Multi-Objective Optimisation

using Sharing in Swarm Optimisation Algorithms

by

Maximino Salazar Lechuga

A thesis submitted to

The University of Birmingham

for the degree of

DOCTOR OF PHILOSOPHY

School of Computer Science

The University of Birmingham

Birmingham B15 2TT

United Kingdom

October 2006

Abstract

Many problems in the real world are multi-objective by nature, this means that many times there is the need to satisfy a problem with more than one goal in mind. These type of problems have been studied by economists, mathematicians, between many more, and recently computer scientists. Computer scientists have been developing novel methods to solve this type of problems with the help of evolutionary computation. Particle Swarm Optimisation (PSO) is a relatively new heuristic that shares some similarities with evolutionary computation techniques, and that recently has been successfully modified to solve multi-objective optimisation problems. In this thesis we first review some of the most relevant work done in the area of PSO and multi-objective optimisation, and then we proceed to develop an heuristic capable to solve this type of problems. An heuristic, which probes to be very competitive when tested over synthetic benchmark functions taken from the specialised literature, and compared against state-of-the-art techniques developed up to this day; we then further extended this heuristic to make it more competitive. Almost at the end of this work we incursion into the area of dynamic multi-objective optimisation, by testing the capabilities and analysing the behaviour of our technique in dynamic environments.

Acknowledgements

First I would like to thank and acknowledge the endless support, advice and motivation that my supervisor Jon Rowe has offered me; without his help, patience and guidance this work would have not been possible.

I would like to acknowledge the support given by CONACyT (Consejo Nacional de Ciencia y Tecnología) through a scholarship to pursue my degree at the University of Birmingham.

I would like to thank to all my family and friends for their constant support.

Special thanks go to my Parents, for their constant and unconditional love and support.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The work in this thesis deals with different concepts, but all of them are bounded or related to one in particular, and that one will be optimisation. This thesis work is about problem solving, and optimisation. Optimisation is the search for the best, not necessarily the global best solution but a good enough solution [61, pg. 79].

In optimisation there are different types of problems, such as linear and non-linear. Linear optimisation problems are not so difficult to solve and there are efficient methods to solve this type of problems (e.g. simplex method) [82]. In this thesis we will focus mainly in non-linear optimisation problems, which are far more complex problems.

Within optimisation, there are also at least four type of problems: single, when we just need to focus on the search of one solution; multi-modal, when there is more than one optimal solution for a problem; multi-objective, when the problem has a set of solutions and we need to find a diverse set; and dynamic problems, where the conditions for a problem change over time and there is need to track the optima. In this work we will be dealing with the last two of them.

1.1 Motivation

Multi-Objective Optimisation Problems (MOPs) can be found everywhere in nature, and we deal with them on a daily basis. When solving MOPs with traditional mathematical programming techniques, they tend to generate a single element of the set of solutions in one run. Moreover, traditional methods are susceptible to the shape or continuity of the set of solutions. Evolutionary Computation (EC) paradigms are very suitable to solve MOPs because of their population-based nature, which can generate a set of solutions in just one run.

The inherent characteristics of evolutionary algorithms make them extremely suitable to solve multi-objective optimisation problems. Due to their use of population of solutions, this type of heuristic is able to give us more than one solution in just one execution, in comparison to classical approaches.

All of the characteristics implicit in EC like: population of solutions and their implicit massive parallel search, their tolerance to noise and abrupt landscapes; make them an excellent tool to be used to tackle MOPs, especially when trying to find several elements of the Pareto front.

Particle Swarm Optimisation (PSO) is a very recent heuristic that solves optimisation problems, it shares many similarities with the way EC heuristics work. Recently PSO has been used to solve multi-objective problems, and has probed to be very effective when compared against EC state-of-the-art heuristics solving MOPs. In our work we will be focusing on the development and improvement of PSO to tackle MOPs.

In particular we feel motivated to do this work, as we want to be able to find evenly dispersed solutions for MOPs (i.e. Pareto fronts with finer and even distributions), in an efficient way (i.e. performing as few as possible fitness function evaluations). As we believe, current techniques don't achieve such quality in the distribution of their solutions over the Pareto front, when performing only a small amount of evaluations to the fitness function.

We also feel motivated to develop a technique, which will be the foundation to an heuristic capable to find well distributed solutions for MOPs in dynamic environments; and as a consequence, to study the performance of PSO in dynamic MOPs.

To the best of our knowledge, an heuristic based on PSO and used to solve dynamic MOPs, has not been studied nor developed until now. And as such, we want to answer the question of: How the PSO technique will perform in dynamic multi-objective environments?

1.2 Contributions

From the research done in this work we expect to acquire more knowledge about particle swarm optimisation when used to solve multi-objective problems; this will be achieved by developing a heuristic capable to deal with them, and by making it highly competitive against algorithms representing the current state-of-the-art in the EC field. We also expect to gain knowledge about PSO when used in dynamic multi-objective environments, and to learn about its capabilities to track dynamic optimum.

In particular the main contributions we expect to give in this work, derived from this thesis, are:

- The development of an algorithm based on PSO capable to deal with MOPs, which gives competitive results when compared against current established state-of-the-art algorithms.
- The development of a mechanism that evenly spreads the solutions found by an heuristic along a Pareto front, when solving MOPs.
- The development of an heuristic capable to successfully track optima in a dynamic multi-objective environment.

1.3 Outline of the Thesis

In Chapter 2, we will start setting the stage of what this thesis will propose. Why it is important to solve multi-objective optimisation problems, why evolutionary methods are suitable heuristics to solve this kind of problems, and why our work is important on these areas. An introduction to multi-objective optimisation is given. It is followed by a description of evolutionary computation, and a review of some of the most relevant evolutionary methods that have been developed to solve multi-objective optimisation problems. We then talk about particle swarm optimisation, and review some of the most important algorithms developed so far that solve multi-objective optimisation problems by using this heuristic. We then review some traditional niching methods used in evolutionary computation. At the end of the chapter, we talk about dynamic optimisation problems and about some of the contributions that particle swarm optimisation has made into this area.

In Chapter 3, we present a particle swarm optimisation algorithm designed to solve multi-objective problems, the algorithm makes use of one of the niching methods described in the previous chapter. This algorithm is then tested with functions taken from the specialised literature, and compared against three other state-of-the-art techniques to solve multi-objective optimisation problems. At the end of the chapter, the findings from the experiments performed are discussed.

In Chapter 4, an enhancement performed to the algorithm presented in the previous chapter is shown. This new algorithm is then tested and compared, and the results discussed.

In Chapter 5, we propose the use of the algorithm presented in Chapter 3 to solve dynamic multi-objective optimisation problems. We present the results we have obtained by testing the heuristic in test problems taken from the specialised literature and discuss the results.

At the end of this document, in Chapter 6 we summarise and give conclusions about the work presented in this thesis, and based on the studies here shown a path for future work is given.

Chapter 2

Background and Review of Related Work

In this Chapter we will introduce some concepts that will be used along this thesis. We will talk about multi-objective optimisation, evolutionary computation, particle swarm optimisation, niching methods and dynamic optimisation.

2.1 Multi-Objective Optimisation

Multi-Objective Optimisation Problems (MOPs) can be found everywhere in nature, and we deal with them on a daily basis. From a person who tries to optimize a budget on a supermarket, trying to get more and better quality products for less amounts of money; industries trying to optimize their production, reducing their production costs and increasing their quality; to people looking for more economical ways to travel and covering bigger

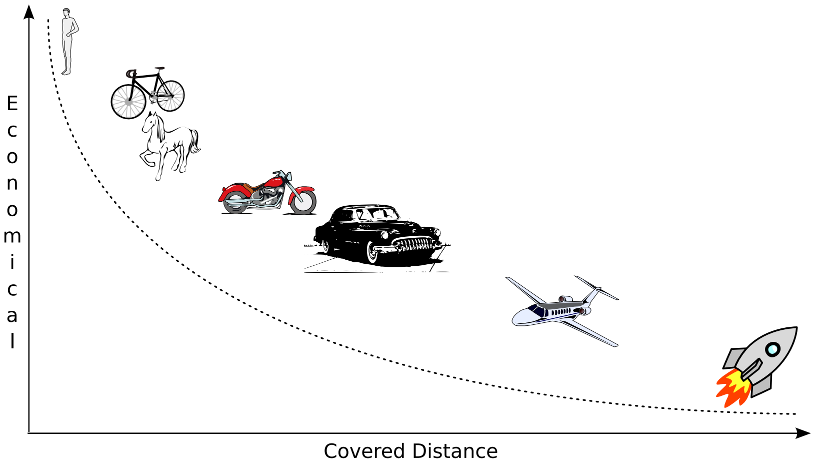


Figure 2.1: In multi-objective optimisation problems there is not a single solution for a given problem. Instead, there is a set of solutions from where we can choose from. Which means of transport we should choose? It might depend on how far we need to go, or how cheap we need it to be.

distances (see Figure 2.1).

Although Multi-Objective Optimisation[1] (MOO) origins are in economics, it has been studied from several disciplines (e.g. game theory, operations research).

In the following sections, we will discuss some historical aspects of MOO. Then, we will talk about some basic concepts of MOO. After that, some classical methods that have been used to solve MOPs will be discussed.

2.1.1 MOO Historical Review

According to Stadler [113, pg. 7], MOO:

> ...is an inherent part of economic equilibrium theory and as such

[1]Some authors prefer to use the term Multi-Criteria Decision Making, Vector Maximum Problem or Vector Optimisation; all of them making reference to the same concept. Here we will use Multi-Objective Optimisation which is the most commonly used term in the EC area.

it can be said to have been founded by Adam Smith in his treatise *The Wealth of Nations* in 1776.

First proposed by Francis Ysidro Edgeworth (1845–1926) and then started to be formalized by Vilfredo Federico Damaso Pareto (1848–1923), MOO has been continuously developed from their origins and is still under research nowadays.

Stadler [112, 113], and Dauer and Stadler [22], are excellent surveys of the MOO history.

Some of the most remarkable points on the origin and development of MOO, as we consider, are:

- MOO origins can be traced back in a book called *Mathematical Psychics* written by the political economist Francis Y. Edgeworth in 1881 [33]. In his book the first mathematical treatment of the MOO problem was given.

- Vilfredo F. D. Pareto in 1906, in his book *Manuale di Economia Politica* [89], extended the concepts of Edgeworth. Pareto in his *Manuale* assumes that every individual seeks to maximize his utility in case nothing stands in his way. One of his more interesting assumptions, and the one that started to establish the concepts of what is now known as *Pareto optimum*, says that, any member of a collectivity is optimum in a certain position if it is impossible to find a way to move it to another position without altering the optimality of the other members.

- Although there is some controversy on who was the real initiator of the game theory area, Emile Borel and John L. von Neumann, both contribute to its inception. At the beginning of the game theory area (years 1921 and 1928), with the introduction of John von Neumann's minimax theorem, MOO also emerged within this independent context.

- In 1950, the phrase *Pareto optimum* was first used by Ian M. D. Little in his book *A Critique of Welfare Economics* [75].

- In 1951, the notion of *efficient point* was first introduced in a paper by Tjalling C. Koopmans [66]. He states: "A possible point in the commodity space is called *efficient* whenever an increase in one of its coordinates can be achieved only at the cost of a decrease in some other coordinate."

- Also in 1951, Harold W. Kuhn and Albert W. Tucker, in his *Non-linear Programming* paper [69], introduced the term *vector maximum problem* and a first formal mathematical treatment of MOO in finite-dimensional spaces was included.

- Leonid Hurwicz in 1958 generalized the previous results of Kuhn and Tucker; and in his paper *Programming in Linear Spaces* [58], he gave the first treatment of MOO in infinite-dimensional spaces.

2.1.2 MOO Basic Concepts

When trying to solve single-objective problems, we only need to focus on the search of a single point in our search space, this is what techniques solving single-objective problems usually do. For MOPs is different, when solving this type of problems what we try to do is to find a set of solutions.

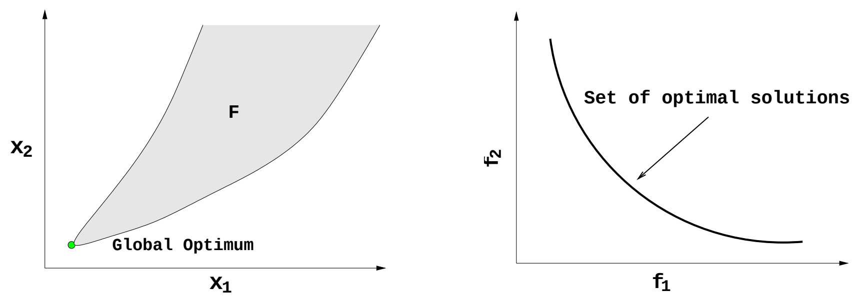


Figure 2.2: Single-objective optimisation vs. multi-objective optimisation.

See left side graph on figure 2.2, the gray area (F) represents all possible solutions to the given problem, and the dot near to the intersection of x_1 and x_2 represents the best solution. That is, we only need to find that single dot. On the other hand (right side graph, figure 2.2), when trying to solve MOPs we are looking not just for one single solution; instead, we are trying to find a set of solutions.

MOPs are not trivial problems to solve and in order to solve them we need to have in mind some factors; as we just saw, we need to find a set of solutions for the problem and we need them as diverse as possible.

When tackling MOPs with traditional mathematical programming techniques, they tend to generate a single element of the set of solutions in one run. Moreover, traditional methods are susceptible to the shape or continuity of the set of solutions. EC paradigms are very suitable to solve MOPs because of their population-based nature, which can generate a set of solutions in just one run.

Formalization Regarding MOPs there are some definitions that formalize the problem. The general definition for a Multi-objective Optimisation

Problem is defined in Coello Coello et al. [21, pg. 6] as:

Definition 1. *Find a vector* $\vec{x}^* = [x_1^*, x_2^*, \ldots, x_n^*]^T$ *satisfying* m *inequality constraints:*

$$g_i(\vec{x}) \geq 0 \qquad i = 1, 2, \ldots, m \tag{2.1}$$

p *equality constraints:*

$$h_i(\vec{x}) = 0 \qquad i = 1, 2, \ldots, p \tag{2.2}$$

and optimizing[2] *the vector function:*

$$\vec{f}(\vec{x}) = [f_1(\vec{x}), f_2(\vec{x}), \ldots, f_k(\vec{x})]^T \tag{2.3}$$

where $\vec{x} = [x_1, x_2, \ldots, x_n]^T$ *is the vector of decision variables.*

In order to optimize a vector function, there is another important concept tied to MOPs called "domination". In Deb [24, pg. 28] the concept of domination is defined as:

Definition 2. *A solution* $\vec{u}$ *is said to dominate another solution* $\vec{v}$*, if conditions 1 and 2 are true:*

1. *The solution* $\vec{u}$ *is no worse than* $\vec{v}$ *in all objectives, or* $f_i(\vec{u}) \ntriangleright f_i(\vec{v})$ *for all* $i = 1, 2, ..., k$.

2. *The solution* $\vec{u}$ *is strictly better than* $\vec{v}$ *in at least one objective, or* $f_{\bar{i}}(\vec{u}) \triangleleft f_{\bar{i}}(\vec{v})$ *for at least one* $\bar{i} \in 1, 2, ..., k$.

[2] The concept of *optimisation* is defined in terms of *domination*, explained later on.

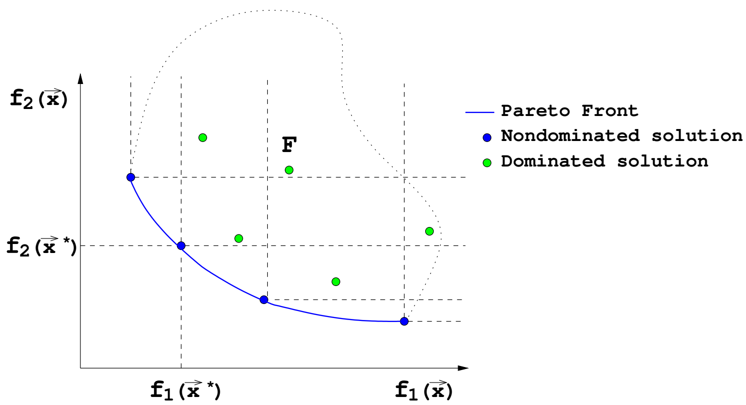


Figure 2.3: An example of non-dominated solutions.

The notation $\lhd$ used, is to express that a solution i is better than a solution j ($i \lhd j$), regardless of the type of problem (minimization or maximization). The notation $\rhd$ is used in the same way, $i \rhd j$ means that solution j is better than solution i.

To exemplify this definition in figure 2.3 we have a minimization problem in both objectives (we are trying to find the points that are closer to the axes). Some dots are representing dominated solutions and some others representing non-dominated solutions. We have four dots along a bold curve and five more randomly placed inside the feasible search space. These four dots along the curve are non-dominated solutions because they are accomplishing both statements above. First, each point is at least not worse than any other for each objective. Second, each point is better in at least one of the objectives. Because between them, they do not dominate each other, they are conforming one set called *non-dominated set*.

Deb [24, pg. 31] defines a *non-dominated set* as follows:

Definition 3. *Among a set of solutions P, the non-dominated set of solutions*

P' are those that are not dominated by any member of the set P.

If within the definition we replace the set of solutions P by the feasible search space F ($P = F$), then the set of solutions in P' will be what is called *Pareto-optimal set* or *Pareto front*.

Again, let us exemplify, in figure 2.3 we have said before that the dots along the bold curve form a *non-dominated set*. Following the above definition we can say that this curve is a *Pareto-optimal set*, because the set of solutions P' along this curve will be all the solutions non-dominated by any member of the feasible search space F.

2.1.3 MOO Classical Approaches

Classical methods to solve MOPs have been in use for at least the last four decades. Because in MOO once that we have solved the problem of searching for solutions, we have to face the decision task of which of all the solutions to chose, this techniques are commonly classified according the way they handle this two problems (the problem of search and make decisions). They are classified as (see figure 2.4) [52, 121, 24, 21]:

- A *Priori*
- A *Posteriori*
- Progressive/Interactive

A detailed review of all the methods mentioned in the following sections can be found in [24, 21, 116].

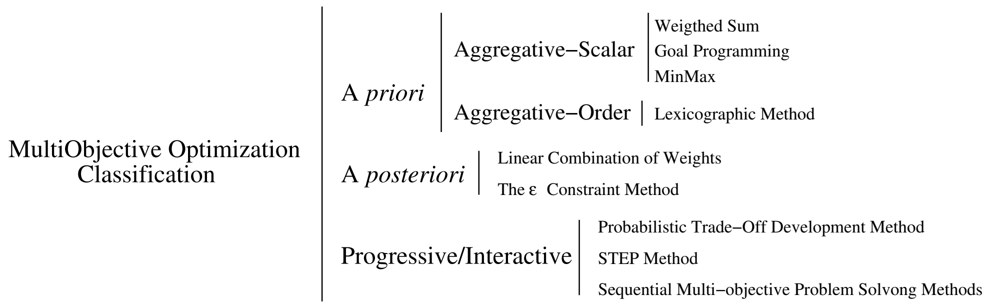


Figure 2.4: Classification of classical multi-objective optimisation techniques.

A *Priori* methods As the name suggests, the first thing to do with this kind of method, before the search for solutions, is to take the decision of which objectives are the most important. Based on that predilection, we can pre-order the objectives and give preference to the objectives chosen by the Decision Maker (DM). A *priori* methods are further divided into *scalar-aggregative* and *order-aggregative.*

Scalar - aggregative approaches combine all the objectives to be evaluated into one scalar-valued utility function. Some of the most representative are:

- Weighted Sum. In this method the DM sets the weights of a weighted sum, like:

$$\min \sum_{i=1}^{k} w_i f_i(\vec{x}) \tag{2.4}$$

  Where w_i ($\in [0, 1]$) is the weight for the i-th objective function. It is a common practice to choose values for the weights that their sum is equal to one. Setting the weights into this method gave to the DM the opportunity to rank the importance of the attributes.

- Goal Programming. This was one of the first methods specifically designed to deal with MOPs. Goal programming involves making trade-offs among the goals until the most satisfying solution is found, which not necessarily means that an optimum solution will be found. The DM has to assign goals for each criteria of the problem, this values are then assigned to the problem as constraints. Then the objective function tries to minimize the absolute deviations from the targets to the objectives. The simplest method is formulated as:

$$\min \sum_{i=1}^{k} |f_i(\vec{x}) - T_i| \tag{2.5}$$

subject to:

$$\vec{x} \in \Omega \ ^3 \tag{2.6}$$

Here T_1 is the target set by the DM for the i-th objective function.

- MinMax. This method seeks to minimize the maximum deviation in any goal to the target. The idea to apply this method to MOPs was taken from game theory which deals with multiple conflicting situations.

Order - aggregative Not all aggregations are scalar, for example:

- Lexicographic Method. In this approach a ranked order of the objectives is given by the DM. Then an optimum solution is obtained minimizing all the objectives, according to the set preferences and starting

[3] Ω represents the feasible region.

with the most important and proceeding with the rest of them.

A *Posteriori* methods The main characteristic of a *posteriori* methods is that they do not need preference of the objectives like a *priori* methods; though, some knowledge on algorithmic parameters could be required in order to find good solutions. With this type of techniques we can obtain portions of the entire Pareto front, with iterating mechanisms. This methods are:

- Linear Combination of Weights. This method uses a weighted sum (see equation (2.4)) as its main mechanism subject to:

$$\vec{x} \in \Omega \tag{2.7}$$

  where $w_i \geq 0$ for all i and positive for at least one objective. Varying gradually the weights w_i from search to search, a sampling of the front can be build up. It is important to note that the values assigned to the w_i coefficient do not reflect the importance of the objectives, they are just factors, when changed, locate different points through the Pareto front.

- The ε-Constraint Method. The characteristic of this technique is to optimize one criterion at a time, keeping one of the objectives and restricting the others within values specified by the DM.

***Progressive/Interactive* methods** The way in which *Progressive/Interactive* methods usually perform is interacting with the DM along the process of optimisation. During the process of optimisation the DM has to give feedback to the technique in order to find (if possible) a better solution. Methods under this classification are becoming very popular in practice due to the interaction they have with the DM, though that is why they lose simplicity. Between the most popular are:

- Probabilistic Trade-Off Development Method. The main characteristic of this method is that the DM interacting with the technique can rank objectives in order of importance at the beginning of the process, and later observe the performance of this preferences to reconcile them with the observed behavior of the attributes. Allowing to the DM not only to interact, but allowing him as well to gain knowledge of the problem.

- STEP Method. The idea of this technique is to find the best solution in a min-max sense; which is in no more than k steps, where k is the number of objectives involved in the problem.

- Sequential Multi-objective Problem Solving Methods. This method takes to the DM in an interactive procedure during the search for a successful course of action.

Classical Approaches (Conclusion) Classical MOO algorithms have some difficulties to solve MOPs. Some of the drawbacks are:

1. Only one solution per run can be found.

2. They have problems when trying to find solutions in non-convex or discontinuous MOPs (see Figure 2.5).

3. Almost all of the classical techniques require knowledge of the problem.

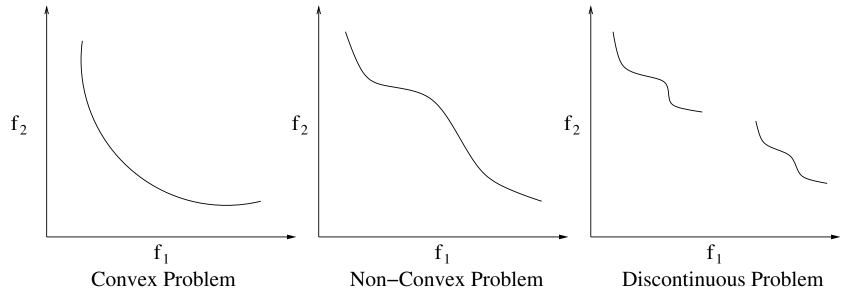


Figure 2.5: Convex and Non-Convex Problems. A convex set is a collection of points such that $\vec{x}_1$ and $\vec{x}_2$ are two points in any position in the set, and the line segment joining them is included in the collection [21].

2.2 Evolutionary Computation

Evolutionary Computation (EC) [4, 48] has its roots on the Darwinian principle of the evolution of the species, biological principles, natural selection mechanisms and genetics. The main idea of natural selection is that the fittest individuals survive through time, and these individuals inherit those characteristics that make them survive to the next generations. Generally, EC paradigms are used as optimisation or search procedures.

Usually the way in which these paradigms work is with the use of populations. These populations are commonly filled by what are known as individuals, which evolve along many generations. In fact, what such individuals represent are possible solutions for one given problem. The role of these individuals is to explore a great amount of different solutions, bounded in a search space, by means of natural selection mechanisms.

EC includes different paradigms. Nevertheless, procedurally almost all EC paradigms follow the same process [61, pg. 143]:

1. Initialize a random population.

2. Evaluate each of the individuals, assigning some fitness.

3. Select the parents for the offspring according to their fitness.

4. Produce the next population applying evolutionary operators to the selected individuals, as crossover and mutation.

5. Return to step 2 until a stop condition is meet[4].

The way in which EC is classified sometimes vary from author to author[5], but generally the majority of the people working in the field divide the EC in at least four main branches [61, pg. 135]:

- Genetic algorithms
- Evolution strategies
- Evolutionary programming
- Genetic programming

As in this work we will only be dealing with the first two, we will not talk about the latter two in the rest of this work.[6]

Genetic Algorithms This paradigm is the most representative of the EC field and one of the most studied as well. One of the first precursors into this area was John Holland, who toghether with his former students in the

[4] Usually, a stop condition is to met a certain number of iterations.

[5] Some authors count genetic programming into genetic algorithms, some others like Heitkötter and Beasley [48] add another branch called "classifier system".

[6] To learn more about evolutionary programming the reader can refer to [41], and for genetic programming to [67].

University of Michigan, made important contributions to the field at the beginning of the 1960s. Genetic algorithms (GAs) are search algorithms, which in a certain level reflect the process of evolution. The way in which this algorithm operates is evaluating a population of individuals, which usually are represented by haploid chromosomes and are codified using bits strings (there also exists implementations with real numbers, integer numbers, characters, etc.). Based on the evaluation of the fitness of each individual, the best ones are selected to generate a new population. In other words, the fittest individuals have a greater possibility to be selected to produce the next generation of individuals. Over this new population, operators of crossover and mutation are applied so it is continuously modified until at least a sub-optimum solution is reached.

Goldberg's [44] book is one of the first of many books on the area explaining the way GAs work. More recent and updated literature can also be found [83, 95].

Evolution Strategies Since the middle of the 1960s, Ingo Rechenberg and Hans-Paul Schwefel experimented with a paradigm that later on will be known as evolution strategies (ES). Until now, this paradigm has had a lot of research activity, especially in Europe.

This branch of the EC shares many characteristics with EP (although both were developed independently for more than 30 years) and GA. Despite the fact that ES makes use of mutation and crossover operators (this last one also known as recombination in ES) they work slightly different from the other techniques.

2.2.1 MOO Evolutionary Computation Approaches

The inherent characteristics of evolutionary algorithms make them extremely suitable to solve multi-objective optimisation problems. Due to their use of population of solutions, this type of heuristic is able to give us more than one solution in just one execution, in comparison to classical approaches.

One of the first suggestions, to use evolutionary approaches to solve MOPs, was made in the late 1950s by G. E. P. Box. He [24, pg. 164]:

> ...hand-simulated an *evolutionary operation* for multiple objectives. He, being an experimentalist, clearly appreciated the necessity of using more than one objective in a design...

Lawrence J. Fogel et al. in 1966 proposed and simulated on a computer a weighted approach to handle multiple objectives, making this application one of the earliest evolutionary algorithms solving MOP [24].

In 1967, Richard S. Rosenberg in his PhD dissertation hinted the use of evolutionary algorithms to solve MOP [44, pg. 199]:

> Rosenberg's study contained a suggestion that would have led to multi-criteria optimisation if he had carried it out as presented. He suggested using multiple *properties* (nearness to some specified chemical composition) in his simulation of the genetics and chemistry of a population of single-celled organisms. His actual implementation contained only a single property, and as a result it can only be considered a hint of things to come.

In 1984, David Schaffer suggested a real implementation of a Multi-Objective Evolutionary Algorithm (MOEA). His implementation is known as VEGA (Vector Evaluated Genetic Algorithm). After Schaffer's work, almost any kind of work regarding MOO was done in the area of Evolutionary Computation (EC).

When Goldberg [44] was describing the use of Genetic Algorithms (GAs) in the MOO area (particularly VEGA, the unique implementation at that time), he realized and described a better implementation for a GA. His suggestion was to use a ranking procedure for *non-dominated* solutions, and a niche and speciation technique to maintain diversity.

After Goldberg's remarks, many researchers recapture the interest for the area and started to implement Goldberg's ideas. The methods developed after Goldberg's suggestions were: multi-objective GAs (MOGAs), niched Pareto GAs (NPGAs) and non-dominated sorting GAs (NSGAs).

There exist a great amount of literature regarding MOEAs, from introductory articles: [52, 19, 17], to detailed surveys about the most transcendental and popular EC methods to solve MOPs. Some works that include comprehensive summaries of MOEAs are: [120, 15, 118, 121, 131]; there are some books on the subject: [24, 21, 88]; and there is as well an Internet repository of material regarding MOEA [16].

2.2.2 Multi-Objective Evolutionary Algorithms

In this section we will describe what we consider some of the most representative multi-objective evolutionary algorithms.

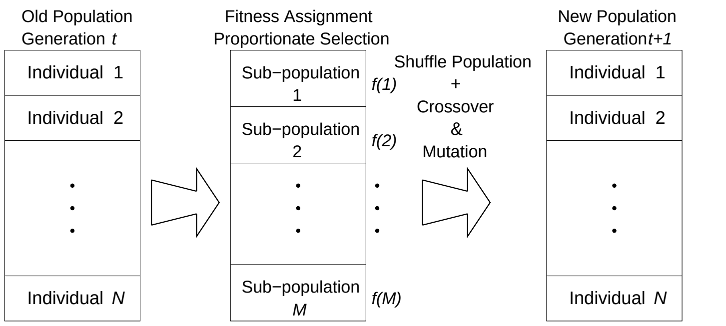


Figure 2.6: Schematic of VEGA, which shows a population of size N being divided in M sub-populations (M being the number of objective functions), to generate a new population.

VEGA

The first evolutionary algorithm employed to solve a multi-objective problem was a genetic algorithm. Schaffer [104] compared his modified GA against an adaptive random technique, and noticed that his new method presented a better performance. His method was called Vector Evaluated Genetic Algorithm (VEGA). VEGA operates by dividing its population into the number of objectives to be optimized (M), then assigns fitness to each individual in every part of the sub-populations just generated, according to each fitness function. Then a proportionate selection is performed inside each sub-population, in order to obtain the best solutions for each objective; as can be appreciated, each sub-population is searching for the best solutions for the objective at hand (see Figure 2.6). After that, crossover and mutation operations are applied over the selected individuals to create a new population.

Due to the nature of this technique, one disadvantage is that because it evaluates individuals against just one objective function (depending in which sub-population they are in), eventually the population will converge into the best solutions for one of those objectives, dismissing or not giving solutions in between. Because of this VEGA has difficulties in finding well spread solutions along the Pareto front.

MOGA

Fonseca and Fleming [42] introduced a Multiple Objective Genetic Algorithm, called MOGA. MOGA assigns fitness to individuals in the population by using a ranking-based fitness method. To assign ranking fitness to a given individual, a rank value of 1 is given and the number of other individuals dominating it is added; (e.g., given a set of solutions, all non-dominated solutions will have a ranking value of 1, and the rest will be ranked according to the number of solutions that they are dominated by; see Figure 2.7).

Due to the use of ranking fitness, all individuals with the same rank share the same fitness value. To evolve an uniformly distributed representation of the global trade-off surface, they incorporated fitness sharing in the objective space between pairwise non-dominated individuals (fitness sharing is described in Section 2.4.1). The use of fitness sharing adds selection pressure to those individuals that reside in poor populated areas, improving the distributed shape of the Pareto front.

It might be considered as a disadvantage to use a fitness sharing mechanism, because the performance is dependent on fixing of a σ_{share} parameter. In their approach [42], introduced a dynamic update of the σ_{share} parameter, which does not need a pre-fixing value for such parameter.

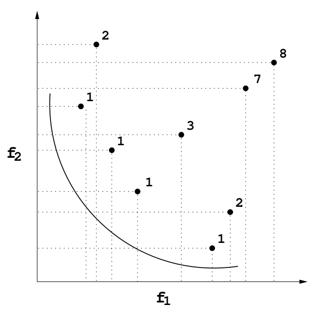


Figure 2.7: MOGA ranks the entire population according to a value given by the number of individuals dominating an individual +1.

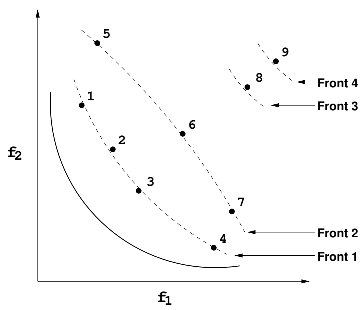


Figure 2.8: NSGA orders the population based in sets of non-dominated fronts. Fittest individuals are those in the closest set to the true Pareto front.

NSGA

In 1994, Srinivas and Deb [110] developed the Non-Dominated Sorting Genetic Algorithm, NSGA. One of the first steps in NSGA is to sort the population in non-dominated sets, individuals belonging to the first non-dominated set are given the highest fitness value; fitness assigned to individuals in subsequent sets is continuously degraded, the further away the set is from the global Pareto front, the smaller the fitness value individuals will receive (see Figure 2.8). With this mechanism NSGA ensures convergence towards the Pareto front.

The sorting mechanism might look similar to the one employed by MOGA, but is not. NSGA groups its solutions by fronts, as can be seen in Figure 2.8 there are four fronts formed; solutions within each front will be assigned the same fitness value. On the other hand, MOGA doesn't group the solutions, instead it just assigns a fitness value according to the number of solutions dominating a given solution, as shown in Figure 2.7.

Because individuals belonging to the same front will have the same fitness values, it is necessary to also ensure diversity. To accomplish that, NSGA makes use of a fitness sharing mechanism that is calculated between individuals for each front; fitness sharing is further explained in Section 2.4.1. This means that, the fitness of each individual belonging to a grouped front will be proportionally decremented according to the number and proximity of other individuals lying within the same front; (e.g., in Figure 2.8, originally all solutions within Front 1 have the same fitness, as they all have the same rank, but then when using fitness sharing the fitness of solution 1, 2 and 3 probably will be not as high as that of solution 4, as this last one is in a less densely populated area than the rest. Although, it is important to note that that wouldn't be the case because NSGA uses fitness sharing in the variable space, rather than in the objective space as MOGA).

This fitness assignment mechanism is then coupled to a GA that uses a roulette-wheel operator to select the parents that will create the offspring.

NPGA

Horn et al. [53] proposed a Niched Pareto Genetic Algorithm (NPGA), which unlike the previous three methods here described, used a binary tournament selection mechanism instead of a proportionate selection.

The binary tournament selection mechanism was modified in two ways; first, *Pareto domination tournaments* were added; and second, when a tournament ends with the two individuals as non-dominant, sharing is implemented to determine a winner.

In the Pareto domination tournaments two candidates for selection are randomly chosen, as well as a comparison set from the parent population. Both candidates are then compared against the individuals in that comparison set; if one of the two is non-dominated by the set, then that one is the one selected. If that is not the case (either, both are dominated or non-dominated by the set), sharing is used to select a winner. Niched counts are calculated for the partially filled next generation population, including both candidates; these are calculated on the objective space. The fittest candidate of the two, which will be the one with a smaller niche count, is the one that will be selected as a winner. The sharing employed was called *equivalence class sharing*.

The described binary tournament selection mechanism is used to select two parents from the parent population which in turn will generate two offspring, and this process will be repeated until a new population is formed.

SPEA and SPEA2

The Strength Pareto Evolutionary Algorithm (SPEA) by Zitzler and Thiele [133], introduced elitism by using an external population of non-dominated solutions at all times.

The algorithm begins with a random generated population P and an empty external population of limited capacity P'. Every iteration, the best non-dominated individuals from P will be copied to P', and P' will be updated. The update is done by eliminating individuals dominated by the new inserted individuals. This mechanism maintains an external non-dominated population with the best solutions found over time. When the capacity of the external population P' is reached, a crowding mechanism is employed to determine which individuals from P' will be retained.

To generate offspring, SPEA assigns fitness to every individual (in both populations) and uses genetic operators to create a new population with a tournament selection mechanism.

SPEA first assigns fitness to solutions in the external population P', this fitness is called strength; the strength of a solution i is determined by:

$$S_i = \frac{n_i}{N+1} \tag{2.8}$$

where n_i is the number of solutions that the i solution dominates in the population P, and N is the size of P.

After assigning fitness to individuals in the external population P', it assigns fitness to all the j solutions in the population P by adding one to the sum of the fitness values of all the members in P' which weakly dominate the individual j, as in:

$$F_j = 1 + \sum_{i \in P' \wedge i \preceq j} S_i \tag{2.9}$$

With this mechanism, solutions in the external population P' that dominate fewer solutions from population P have better fitness; and solutions in population P, which are being less dominated from solutions in the external archive P' have better fitness (see Figure 2.9).

Zitzler et al. [132] presented an update to SPEA called SPEA2 [132]. The main differences from the previous approach are: it used an improved fitness scheme, a nearest neighbor density estimation technique was incorporated, and it used a new archive truncation method.

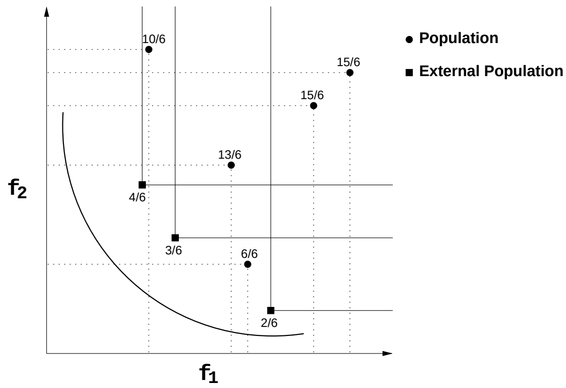


Figure 2.9: SPEA fitness assignment.

PAES

Knowles and Corne [62, 63] presented a MOEA that makes use of an evolution strategy, called the Pareto-Archived Evolution Strategy (PAES).

The algorithm may be identified as a $(1+1)$ evolution strategy; it begins by generating a random solution which is evaluated and added to an external archive (see Algorithm 1, lines two to four). The solution is copied, the copy is mutated and evaluated; when the mutated-copy represents a better solution (in terms of Pareto dominance), than at least one solution from those stored in the archive, solutions dominated by the new individual will be deleted from the external archive, and the new solution will be inserted into this archive (lines six to thirteen, Algorithm 1).

In the case where all the solutions in the repository are as equally good as the solution just generated, an innovative density calculation procedure is used to aid on the decision of which solutions to retain. This mechanism uses hypercubes, which are generated by a conceptual grid set in the objective space. The idea of hypercubes is to distinguish between crowded and poor populated areas, which is why all the current solutions are located within the hypercubes.

In this way solutions which reside in hypercubes less populated, will have a better chance to be selected as a parent to generate another solution in the next generation (lines eighteen and twenty-five, Algorithm 1). And solutions in more dense hypercubes will always be prone to be selected when in need of deleting an element from the repository (line twenty-three, Algorithm 1).

NSGA-II

Deb et al. [25, 27] developed an elitist non-dominated sorting genetic algorithm called NSGA-II.

NSGA-II starts by creating a random population P and an offspring population Q; the offspring population is created by using a crowded tournament selection, crossover and mutation operators. Then both populations are joined and sorted, in order to select the solutions that will form the population for the next generation (see Figure 2.10 and Algorithm 2).

NSGA-II main mechanism is a non-dominated crowding sort process; this process starts by sorting the individuals in the population in a similar way to NSGA, where all the solutions will be ranked according to the closeness to the Pareto front (see Figure 2.8). Then a crowding distance assignment procedure is performed over the individuals in each one of the fronts formed. The crowding distance value assigned to an individual is proportional to the perimeter of a cuboid formed by its nearest neighbors in the same front. A bigger value will mean a less crowded area for a given solution.

NSGA-II then uses this sorting mechanism to select the solutions that

Algorithm 1: Pareto-Archived Evolution Strategy (PAES).

```
begin
    Create Random(x)
    Evaluate(x)
    Add to Archive(x, A)
    while stop criteria not satisfied do
        x' ← Mutate(x)
        Evaluate(x')
        if Dominated by Any(x', A) then          /* When x' is
        dominated, a new solution will be generated using x
        */
            x ← x
        else if Dominated Any(x', A) then   /* When x' dominates
        at least another solution in A */
            Update Archive(A, x')
            Add to Archive(x', A)
            x ← x'
        else         /* When all solutions are equally good */
            Calculate Hypercubes(A, x')
            if A < MaxArchiveSize then
                Add to Archive(x', A)
                x ← Winner(x, x')
            else
                if Highest Hypercube(x') then
                    x ← x
                else
                    Delete Element Highest Hypercube(A)
                    Add to Archive(x', A)
                    x ← Winner(x, x')
                end
            end

    end
end
```

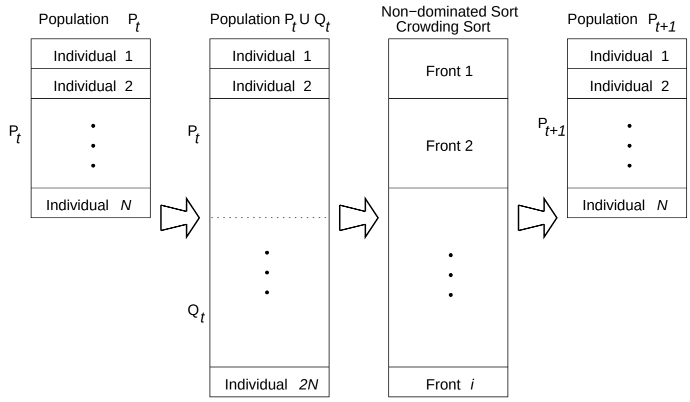


Figure 2.10: Schematic of the NSGA-II procedure. First, offspring population Q_t is created from P_t, using crowded tournament selection, crossover and mutation operators; and combined to form a population of size $2N$. Then, population $P_t \cup Q_t$ is sorted with the crowding-sort mechanism. Finally, population P_{t+1} is created.

will be part of the population in the next iteration. Individuals belonging to fronts closer to the real Pareto front will always have a higher priority rather than those that doesn't, but when two individuals are positioned in the same front, the crowding factor will help to decide for a winner.

Algorithm 2: Non-dominated Sorting Genetic Algorithm (NSGA-II).

```
begin
    P ← Create Random(N)
    repeat
        Q ← Create Offspring(P)
        R ← P ∪ Q
        Non-dominated Sort(R)
        Crowding Sort(R)
        P ← Select Solutions(R, N)
    until stop criteria not satisfied
end
```

Evolutionary Approaches (Conclusion) The nature of evolutionary techniques give them some advantages over the classical methods. Some of them are:

1. Multiple solutions per run can be found.
2. They are not susceptible to non-convex or discontinuous MOPs (see Figure 2.5).
3. No previous knowledge of the problem is required.

All of the characteristics implicit in EC like: population of solutions and their implicit massive parallel search, their tolerance to noise and abrupt landscapes; make them an excellent tool to be used to tackle MOPs, especially when trying to find several elements of the Pareto front.

2.3 Particle Swarm Optimisation

A relatively new optimisation technique called particle swarm optimisation has been widely studied and applied to solve optimisation problems [32, 34]. In this section we will introduce this technique, and then we will discuss how it has been modified to handle problems with multiple objectives.

2.3.1 PSO

Developed by Kennedy and Eberhart [60], particle swarm optimisation (PSO) was inspired by the way birds flock. The main concept resembles the way birds travel when trying to find sources of food, or similarly the way a fish school will move when approaching sources of food.

The idea behind it, is that a bird in a flock, or a fish in a school, will always be guided by the rest of the birds, and birds will also get influenced by that bird—depending on the performance of the bird. This is a model where everybody gets influenced by the experience acquired by each member of this "society"; and by this interaction, where members share their experiences, an emergent behavior of search towards an objective occurs—in the case of the birds, a source of food. PSO makes an analogy to the birds' behavior just described. In PSO, "particles" parallels birds, and the "swarm" resemblances a flock.

The way this behavior has been adapted to solve optimisation problems is: the positions of particles, inside the swarm (or population), are treated as solutions to a given problem; then this particles move or travel through the search space—looking for new and/or better solutions for the problem—, during the process they share their experiences with the rest to accomplish the objective of finding the best solution for the problem.

Mechanics

Particles in the swarm travel or move in the search space following two points: (a) its memory, which is the best location a particle has achieved so far *pbest*; and (b) a leader in the swarm, which is chosen according to the best solution found so far by the swarm or a portion of the swarm (global best (*gbest*) or local best (*lbest*) topologies[7]).

A swarm is formed by particles which are defined as multidimensional points. Therefore, in a j dimensional search space, a particle i is a j-

[7] This topologies will be described further on.

dimensional vector

$$\vec{x_i} = (x_{i1}, x_{i2}, \ldots, x_{ij}).$$

The velocity of every particle i is also a j-dimensional vector

$$\vec{v_i} = (v_{i1}, v_{i2}, \ldots, v_{ij}).$$

The personal best position of each particle i is a point, denoted by

$$\vec{pbest_i} = (pbest_{i1}, pbest_{i2}, \ldots, pbest_{ij}),$$

as it is the global best position in the swarm, denoted by

$$\vec{gbest_i} = (gbest_{i1}, gbest_{i2}, \ldots, gbest_{ij}).$$

A particle, to update its position in the search space, adds a velocity value to its current position. Like in the following equation:

$$x_{ij}^{t+1} = x_{ij}^{t} + v_{ij}^{t+1} \tag{2.10}$$

where a particle i, to update its position x at time $t+1$, adds a velocity v value (updated for time $t+1$) to the particle's position at time t, for every j dimension.

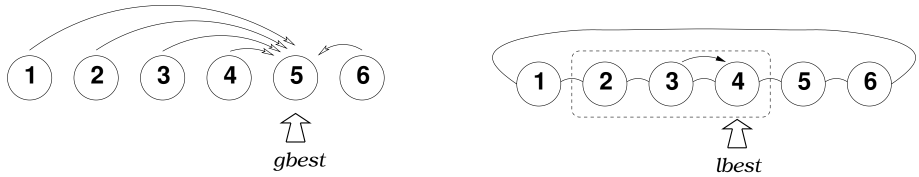
Originally, the velocity for a particle i for each j dimension at time $t+1$, was updated according to the following equation:

$$v_{ij}^{t+1} = v_{ij}^{t} + c \times r_{1j} \times (pbest_{ij}^{t} - x_{ij}^{t}) + c \times r_{2j} \times (gbest_{ij}^{t} - x_{ij}^{t}) \tag{2.11}$$

where v_{ij}^{t} is the value of the velocity at time t for the particle i, c is a positive constant, r_{1j} and r_{2j} take independent uniform random values in the range $[0, 1]$, x_{ij}^{t} is the position of the particle i at time t, $pbest_{ij}^{t}$ is the best personal position found by a particle i up to the time step t, and $gbest_{ij}^{t}$ is the best position found so far by the swarm up to time t (where the best position will be defined by the topology in use, *gbest* or *lbest*).

Topology

Originally, PSO has mainly two topology models *gbest* and *lbest*. In the *gbest* model, all particles in the swarm follow a global leader during the optimisation process; in other words, the search for the optimum will be guided by the particle that has found the best solution at a given time (see Figure 2.11a). On the other hand, the *lbest* model, doesn't have a single leader for all the particles in the swarm, each particle has to select a guide. In this model particles are connected with their nearest neighbors and between them they select which one has the local best optimum value *lbest*, which will be the one selected to guide the search. Although the neighborhood can be defined in multiple ways, typically this model selects a *lbest* for each particle i by comparing the particle i and its two neighbors, $i-1$ and $i+1$; the best



(a) In the *gbest* model, all particles adapt their velocity values according to the best global position found by the swarm. In this example, the best global position is particle 5, which would be used as *gbest* when updating velocities with Equation 2.11.

(b) In a *lbest* model, particles have connections with other particles physically adjacent to them. For example, when selecting a leader for particle 3, its closest two neighbors are particle 2 and 4; for this example, the fittest from these three (particle 2, 3 and 4) is 4, and is the one to be used as *lbest* to update the velocity of particle 3 in Equation 2.11.

Figure 2.11: Graphical example of (a) *gbest* and (b) *lbest* topologies. Here a swarm of six particles is used to show their characteristics.

one within the neighborhood is the one that will be selected as a leader (see Figure 2.11b).

Inertia Weight

The equation used to update the velocity of particles, as shown in Equation 2.11, has been slightly modified since its inception in [60]. A more widely used formula is the one introduced by Shi and Eberhart [106, 107], where an inertia weight w is added to Equation 2.11.

$$v_{ij}^{t+1} = w \times v_{ij}^{t} + c_1 \times r_{1j} \times (pbest_{ij}^{t} - x_{ij}^{t}) + c_2 \times r_{2j} \times (gbest_{ij}^{t} - x_{ij}^{t}) \tag{2.12}$$

where c_1 and c_2 are positive constants, referred as acceleration coefficients or cognitive and social parameters, respectively. The motivation for them to introduce inertia weight w into the equation is to balance the two main parts in the formula, the cognitive part and the social part.

Equation 2.11 can be seen as a compound of three parts; the first part refers to the previous velocity, which gives impetus to the particle; the second part or cognitive component, is the personal knowledge of a particle, which motivates the particle to move towards its best known position; the third part or social component, represents the collective effort of the particles to find global optima, this component will move the particle towards a global best position.

Because the cognitive part in the formula is related to the local search ability of a particle (*pbest*) and the social part to its global search ability (*gbest*), the inertia weight w is meant to regulate the trade-off between the two. Large values of w encourage the particle to roam in bigger step sizes, favouring global exploration; whereas smaller w values force the particle to explore in smaller step sizes, favouring local exploration.

Similar to the temperature adjustment schedule found in Simulated Annealing algorithms, inertia weight w has shown to give better results when linearly decreased over time during the course of a run [106, 107]. This is due to fact that at the beginning of the run particles will be given the priority to search for global best positions, and towards the end of the run their priority will be to fine tune the best positions previously found.

It is worth mentioning, that there exist different formulations for the calculation of the velocity for PSO, a discussion of some basic variations can be found in [34, Chap. 12.3]. In the work presented in this thesis we adopted the inertia weight model, as it was the one that we found showed better results in our extensive set of experiments. Although a proper study was not performed to support this conclusion.

Algorithm

The PSO algorithm has a cycle run similar to that of evolutionary computation paradigms (see Section 2.2). An outline for the PSO algorithm is given in Algorithm 3.

First, a swarm of M particles is randomly generated, evaluated, and their memories and velocities initialised (see lines two to seven, Algorithm 3). The memory in each particle is usually initialised with the same value that the particle initially holds; the velocity for each particle is commonly initialised with a value of 0.

After initialisation has taken place, the memory in each particle is updated. When a particle's position is in a better location than the one stored in its memory, this last one will be replaced by its current position (lines ten to twelve, Algorithm 3). Updating the global best position *gbest* will also take place; whenever there is a global best position found, the global best will be updated accordingly (lines thirteen to fifteen, Algorithm 3). New velocity and position values for particles will be calculated for each particle according to Equations 2.12 and 2.10 (lines seventeen to twenty, Algorithm 3), respectively. This process (lines eight to twenty-one, Algorithm 3) will be repeated until a stop criteria has been met, which usually is to perform a certain amount of iterations.

Algorithm 3: Particle Swarm Optimisation (PSO).

```
begin
    for i ← 1 to M do
        Initialize Particles(x_i)
        Evaluate(x_i)
        Initialize Memory(pbest_i)
        Initialize Velocities(v_i)
    end
    repeat
        for i ← 1 to M do
            if x_i better than pbest_i then    /* update memories */
                pbest_i ← x_i
            end
            if pbest_i better than gbest_i then    /* update global
            best */
                gbest_i ← pbest_i
            end
        end
        for i ← 1 to M do
            Calculate Velocities(v_i)
            Update Position(x_i)
        end
    until stop criteria not satisfied
end
```

Discussion

Previously, Kennedy and Eberhart [61] have mentioned that PSO shares many similarities with EC paradigms, and that PSO has been influenced by EC since its inception, and still is; and that PSO has even been considered a fifth component of the EC area. A more recent article by Engelbrecht [35], discusses the similarities and differences between the different EC paradigms and PSO. The article to an extent concludes that the characteristics that the PSO technique shares with the EC paradigms are mainly "cosmetic", and PSO should be classified within the swarm intelligence paradigms.

Although, some debate still exists about classifying PSO as an evolutionary computation or a swarm intelligent paradigm. One reason why PSO is attractive to solve optimisation problems, is that it shares some similarities with the way some of the EC heuristics work (as the ones referred in Section 2.2). For that reason, PSO has been employed to solve the type of problems that we deal within this thesis.

2.3.2 Particle Swarm Optimisation on MOO

EAs are capable of solving MOPs providing acceptable results. As we have previously discussed, PSO shares some characteristics with EC methods. Particle swarm optimizers have been successfully used to solve non-linear single-objective optimisation problems, and they have been widely compared against EC methods [2, 31, 122]. Because of this reason it was a natural step for PSO to be extended to deal with MOPs. Some of these extensions have been summarized in [39, 56, 109, 99]. In this section we will mention some of the extentions and modifications that have been done to the PSO in order to make it able to solve MOPs.

Moore and Chapman (1999) One of the first attempts to make PSO to handle MOPs was done by Moore and Chapman [84]. In this unpublished document they modified the way particles in the swarm use their personal best experience to move towards a global optimum. As we have seen particles in the swarm use two points as a reference towards where to move in the search space, their own best position (*pbest*) and their neighbor best position (*lbest*)[8]. Particles instead of keeping track of one personal best position (*pbest*), they maintain a list of previous non-dominated solutions (*pbests*). To calculate new velocities, two pieces of information are needed, the best position discovered by a particle (*pbests*) and the best position discovered in the neighborhood (*lbest*). When a particle has two or more *pbests* in their list of non-dominated solutions found so far, they choose one of those at random, and the *lbest* is chosen after finding a non-dominated solution within the neighborhood.

Ray and Liew (2002) Another extension to PSO to deal with MOPs was made by Ray and Liew [94]. To select the leaders guiding the search of the particles, they made use of a Pareto ranking scheme, crowding radius and roulette wheel. First they ranked the particles in the population according to dominance, particles non-dominated in the swarm had the highest rank,

[8]We have to remember that in PSO there are two main connection models between particles, *gbest* and *lbest*. In this case Moore and Chapman use the *lbest* model where a local best is followed, instead the *gbest* model where a global best is followed.

these particles were preserved as a set of leaders. Then for each of the leaders a crowding radius based in the objective space was computed. And finally to ensure that leaders with less crowding density were chosen, a roulette wheel scheme was used to select a particle's leader.

Coello and Salazar (2002) Multi-Objective PSO (MOPSO) by Coello Coello and Salazar Lechuga [18] is another of the first extensions made to PSO to solve MOPs. This heuristic made use of an external repository to store the non-dominated solutions found during its execution. Particles were guided towards the Pareto front using those in the repository as leaders, and to maintain diversity the concept of a dynamic grid, similar to the one in PAES, was used. This grid is employed to spread particles along the Pareto front and avoid overcrowded areas. The proposal was further studied in [20].

Hu and Eberhart (2002) At the same time, Hu and Eberhart [54] proposed another extension to the PSO to handle multiple objectives. Because in its original form PSO is restricted to find a single solution, this limits PSO when trying to find more than one solution when we have at least two objectives. To overcome this problem, in this technique they fixed the fitness values in one objective and concentrate on the optimisation of the other objective (in the case of two-objective problems). To optimize it they made use of a dynamic neighborhood mechanism. The way this mechanism works is by calculating distances between all the particles in the objective space, then each particle find its nearest neighbors based on that distance, and finds the local optimum among them. This optima will act as the *lbest* for that particle. The *pbest* of a particle is updated only when a new solution dominates its current *pbest*.

Parsopoulos and Vrahatis (2002) Parsopoulos and Vrahatis [91, 92] modified PSO to deal with MOPs by using a weighted aggregation approach, as this is one of the most common classical approaches to solve MOPs. According to this all the objectives have to be added to a weighted combination. They experimented with the following approaches: *Conventional Weighted Aggregation* (CWA), were all the weights are fixed; *Bang-Bang Weighted Aggregation* (BWA), were the weights are abruptly changed during optimisation; and the *Dynamic Weighted Aggregation* (DWA), were the weights are gradually modified.

In the same study, they also experimented borrowing concepts from Schaffer's VEGA (Vector Evaluated Genetic Algorithm) [104], and created what they called VEPSO. In a two-dimensional problem, VEPSO uses two swarms, one swarm to evaluate each objective. Information to calculate velocities is shared between swarms, the best (or bests) particle(s) from one swarm is/are used to calculate the new velocities on the other swarm.

Fieldsen and Singh (2002) In [38], they introduced a method that utilizes the *dominated tree* data structure [40], which in essence is an elitist unconstrained archive. With this structure, a global set of non-dominated solutions is maintained. A set of best solutions for each member in the swarm is also kept. According to those two sets a *gbest* and *pbest* are then selected to calculate new velocities for the particles.

In that study, they also introduced an stochastic variable named "turbulence". This variable acts as a mutation mechanism within the velocity formula of the original PSO, and when used a significant increase of performance was claimed.

Bartz-Beielstein et. al (2003) Bartz-Beielstein et al. [5], used an external archive to maintain the best solutions found by the particles during their fly experience; and proposed a technique to select the *gbest* and *pbest* particles to calculate velocities, and to select members of an archive to be deleted when it becomes overcrowded. To achieve this, they proposed six functions for selection of members, the functions to use are selected by the user. When a *gbest* or *pbest* is needed an archive member will be selected according to the fitness given by one of three functions. A similar situation occurs when the archive size is surpassed, a member is eliminated according to the fitness given to the archive members by one of three functions. These functions mainly assign their fitness according to clustering techniques and to success-based selection.

Hu et al. (2003) In [55], Hu et al. presented a modification to their former dynamic neighborhood PSO [54]. They extended their DNPSO by introducing an external archive to store non-dominated solutions, the external archive is referred as an "extended memory", similar to the repository used by [18]. In their previous approach, they based their optimisation search process on neighborhood and personal experiences. With this modification the optimisation process now relies more on the extended memory.

Li (2003) In [72], Li proposed the non-dominated sorting PSO (NSPSO). His proposal borrowed concepts from NSGA II [25], and a widely used niching method [53]. In this proposal, particles are pressured to move towards the Pareto front by combining their personal best positions and the offspring of the swarm. This combination generates a population double in size, which is then truncated by using a non-dominated ranking system similar to the one presented in NSGA II; by selecting the foremost particles in relation to the Pareto front. To maintain diversity, niching methods previously introduced by Goldberg and Richardson [45] are used; a niche count (parameter free) proposed by Fonseca and Fleming [42], and a crowding distance previously proposed by Deb et al. [25].

Mostaghim and Teich (2003) In a paper by Mostaghim and Teich [86] a new method to find guides for particles towards the Pareto front is introduced. This method is called the Sigma method. In this implementation they use an elitist external archive, and they choose leaders for the particles to follow from that archive. To select a leader for a particle, from the archive, they use the sigma method, which calculates a sigma factor for each particle in the external archive and a sigma factor for each particle in the swarm. Then for each particle in the swarm they estimate the distance between its sigma factor and each one on the archive, to select from the archive the one with the smallest distance value. In this way they made the particles move towards the Pareto front. This method employed the turbulence factor as previously used by Fieldsend and Singh [38] in their multi-objective PSO adaptation.

Later, Mostaghim and Teich [85], compared the use of ϵ-dominance against a clustering technique in MOPSO. ϵ-dominance is used to bound the size of non-dominated solutions in an external archive, as opposed to clustering. They found that the ϵ-dominance method can find results much faster than clustering and in some cases it can find better or comparable results. They also suggested the use of an initial external archive with diverse solutions, rather than an empty external archive; as this influences the diversity of solutions, particularly at the start. In the same study they also presented the Sigma diversity metric, which is based on their previous work [86].

Lu (2003) In [76], Lu reported two approaches to solve MOPs with the use of PSO; one called Dynamic Particle Swarm Optimisation (DPSMO), and another one called Dynamic Particle Swarm Evolutionary Algorithm (DPSEA). DPSMO was developed by using a dynamic population strategy [77]. The implementation was similar to the dynamic multi-objective evolutionary algorithm (DMOEA) proposed in his previous work [77]; where they proposed an evolutionary algorithm that uses a population of growth and decline strategy. The growth and decline of the population size in the strategy is defined by Pareto ranking, age and density indicators. In this proposal instead of using an evolutionary algorithm they made use of PSO. Lu [76] reported better results when compared against their DMOEA, in terms of faster convergence and better distributed Pareto fronts. In the same study, DPSEA was developed which is mainly based in DPSMO but adding a crossover operation borrowed from DMOEA.

Srinivasan and Seow (2003) Srinivasan and Seow [111] created a hybrid algorithm combining concepts of PSO and an evolutionary algorithm called PS-EA. The main mechanism for this method is called SUM (Self-Updating Mechanism), which uses a Probability Inheritance Tree (PIT). The SUM is inspired by PSO concepts, and works as an emulator of the different positions that a particle could adopt when using PSO equations to update its current position. The mechanism will update the current position of a particle to the next, according to the result that lies at the bottom of the probability inheritance tree. The results lying at the bottom of the branches, which are used to update particle's positions, were obtained by analyzing the original PSO formula. The branch to choose whitin the tree will depend on the probabilities given by a dynamic inheritance probabilistic adjuster, introduced in the same study. Although, the study is more focused on showing the performance comparison between a genetic algorithm and PSO schemes, in multi-modal single-objective optimisation; they present a multi-objective case, which shows the applicability of this heuristic to solve MOPs.

Zhang et al. (2003) Although the proposal by Zhang et al. [126] is not very clear, it is assumed that they proposed to solve MOPs by solving each objective using a swarm on every objective. The *gbest* value that is used to calculate the velocity by the particles, to update their positions, is an average value of all the *gbest* in each swarm; and the *pbest* value used, is an average of the *pbest* or by randomly selecting one.

Mostaghim and Teich (2004) In [87], they present a new MOPSO called *covering* MOPSO. This work makes use of their previous studies [86, 85]. In this paper what they try to do is to improve the quality of the Pareto front, by covering the gaps that might been left; usually by the majority of the solutions found by any MOEA or MOPSO, using an external archive limited in size. To accomplish this, they first make an initial run with an algorithm similar to the one presented in [85], to get an approximation to the Pareto front. Then with the solutions obtained from that initial run, sub-swarms are created according to the Sigma method presented in [86], and the particles explore the areas around the initial non-dominated solutions. Hence closing the gaps when finding more solutions belonging to the Pareto, and incrementing the number of non-dominated solutions from the initial archive. The size of the external archive for this method is unlimited.

Li (2004) In [73], Li presented the *maximin*PSO. This model presents a similar approach to the previously presented NSPSO [72], but in this new proposal Li uses the maximin fitness function to rank particles in the swarm, rather than the non-dominated sorting procedure previously used. With the maximin fitness function evaluation, is easy to compare solutions by using the fitness value assigned to each decision vector in the population. In general, when solutions have a fitness value less than zero they are non-dominated, when they have a value equal to zero they are weakly-dominated, and when they have a fitness value greater than zero they are dominated solutions. Besides facilitating dominance comparisons, diversity maintenance is also accomplished through the use of the maximin fitness function. Solutions that are in less crowded areas tend to present a better fitness function value. This eliminates the use of niching methods previously employed in [72]. This technique shows good results when compared against NSGA-II [25].

Mahfouf et al. (2004) In this approach Mahfouf et al. [79], proposed a modification to the original equation of the PSO to calculate velocities. They called this approach Adaptive Weighted PSO (AWPSO). In this proposal, an acceleration factor was added to the original velocity equation; the value of this acceleration factor increases as the number of iterations increases. This will enhance the global search at the end of the run to jump out of local optimum, as they claim. Also a random uniformly distributed factor is used, this factor makes the weight of the original velocity equation, to vary randomly between $[0, 1]$, like the one used by Zhang et al. [125]. In this approach instead of using a dominance based evaluation, a weighted aggregation approach is used in order to evaluate particles. A non-dominated sorting technique is also used to identify the best solutions in each generation.

Baumgartner et al. (2004) In this approach [6], the fitness of each particle is calculated according to a weighted aggregation function. The swarm is partitioned into n equally sub-swarms, and each sub-swarm is guided by the particle with the best weight aggregated value.

Parsopoulos et al. (2004) Based on their previous work, in this study Parsopoulos et al. [90] proposed a parallel version of VEPSO [92]. In VEPSO, more than two swarms are used to optimize each of the objectives that the problem at hand involves. Each swarm is guided according to the best solutions found by other particles belonging to a different swarm. The procedure of exchange of information between swarms can be seen as a migration scheme. In this study they mention two ways of selecting a swarm to select a *gbest*: one by using a ring migration topology, where adjacent swarms interchange information; and another by random selection.

Toscano and Coello (2004) Toscano Pulido and Coello Coello [115] proposed a method called "Another Multi-Objective Particle Swarm Optimisation" (AMOPSO). The novelty in this approach is the use of a clustering technique to divide the particles in the swarm, creating several sub-swarms, in order to obtain a better distribution of solutions in the decision space. This can roughly be described as if different PSOs were concurrently solving different problems (optimizing a particular area in the search space), and from time to time exchanging information between them; as they will exchange some of their best solutions found after an user specified period of time. The best solutions on each swarm are selected using a pareto ranking scheme.

Chow and Tsui (2004) In [14], a method called "Multi-Species PSO" (MS-PSO) is introduced. This method uses a set of swarms to solve MOPs, this set is equal to the number of objectives to be optimized. In addition, the method proposed a modification to the way PSO influences the flight of the particles. Beside particles being attracted towards their best personal experience and the best global experience from the swarm, they introduced a new factor in the PSO velocity equation; this factor influences the flight of the particles by being attracted towards the best particle in a neighbor swarm.

Alvarez et al. (2005) In this article by Alvarez-Benitez et al. [1], three methods to select particle global guides are proposed. This methods are exclusively based in dominance; no distance methods objective-space-based to maintain diversity are used, as in the majority of methods previously proposed. This methods are: *Rounds*, which promotes diversity in the population; *random*, which promotes convergence; and *prob*, which is a weighed probabilistic method that exhibits a compromise between the previous two. They also studied four methods in which particles are confined into the feasible region of a problem, and showed the repercussion that the method in use can have in the overall process of optimisation. It was found that the use of a method that allows particles to stay close to the boundaries of the feasible region enables a faster location of the Pareto front. This method also makes use of a non-limited in size external archive, to store the non-dominated solutions found; and makes use of a perturbation factor in the velocity formula.

Villalobos et al.(2005) Villalobos-Arias et al. [123] proposed a method to promote diversity, which is based on the use of what they called "stripes". This method is tested by pairing it to a previously developed MOPSO [20]. They define stripes as lines that divide the line between the two more distant solutions, in the case of a two objective problem (although it can be generalized for more than two objectives). Along that line, several uniformly distributed points lie, these points are called stripe centres. Particles from the swarm guide their search taking as guides these stripe centres.

Reyes and Coello (2005) In [97], a new multi-objective PSO algorithm is proposed. The new algorithm uses Pareto dominance and a crowding selection mechanism to remove leaders from over-crowded areas when the external archive, used by the algorithm to store its best solutions, has reached its limit size. Selection of the leaders is based on crowding fitness by means of a binary tournament. The use of mutation operators is introduced, mutation operators known in the EA literature as: uniform mutation and non-uniform mutation. To use this mutation operators they proposed a scheme where the swarm is subdivided in three; two of these sub-swarms use one of these operators each to modify the values of the decision variables, and a third sub-swarm does not make use of mutation at all. To fix the size of the external archive the concept of ϵ-dominance is adopted.

In a different study, Reyes-Sierra and Coello Coello [96], incorporated the concept of fitness inheritance into their previously proposed MOPSO [97], in order to reduce the number of function evaluations performed. The inherit fitness of a particle is calculated using the previous position of the particle and its leader. Particles with inherited values can not enter the final Pareto front.

In a similar study [98], they also introduced the use of approximation techniques in addition to fitness inheritance. Fifteen variants of fitness inheritance and three of fitness approximation are presented. In the study, the approximation technique adopted was based on the objective values of the closest neighbors.

Ho et al. (2005) Ho et al. [49] proposed an extension to the original PSO, the following are the main characteristics of this algorithm: 1) use an external archive to store Pareto front solutions; 2) use of a mechanism to assign fitness similar to the one proposed in [135] for the SPEA algorithm; 3) to select a personal best, all the particles maintain track of all the non-dominated solutions found by each of them, and an age variable is included to each of this personal bests positions. To select a *pbest* from this personal memory, weighted sums of their age and fitness values are assigned to each personal best stored solution, and then a Roulette Wheel selection mechanism is employed; and 4) selection of *gbest* is similar to the process of the selection a *pbest*, except that for a global leader the entire archive of non-dominated solutions is used.

Janson and Merkle (2005) ClustMPSO, developed by Janson and Merkle [59], is an approach that uses multiple swarms. The K-means clustering algorithm is used to separate particles into several swarms. Each of these swarms has its own non-dominated front, these fronts are independent, as they might be dominated by solutions in other swarms. To update the *pbest* position of a particle two options are given: 1) when a solution is dominated the position will be updated, or if both solutions are mutually non-dominated a probability is considered in order to update. 2) A weighted sum of the function values for the problem is calculated, and the position with the best value is retained as personal best. The non-dominated front in each sub-swarm is used to select the *lbest* for a particle to adapt its velocity. A particle will select a *lbest* inside the sub-swarm and will stick to it for a given number or generations or until it drops out from the non-dominated front. If a swarm becomes dominated (none of its particles is in the total non-dominated front), the swarm is reset randomly with values taken from another swarm.

Zhao and Cao (2005) The adaptation by Zhao and Cao [128], shares many similarities to the one proposed before by Coello Coello and Salazar Lechuga [18]. It employs an external archive to maintain its non-dominated solutions, and uses a geographically based approach to maintain diversity of solutions in the archive (similar to the grid mechanism used in PAES [62]).

Zhang et al.(2005) Zhang et al. [127] and Meng et al. [81], made a proposal called Intelligent PSO (IPSO) to solve MOPs. In this proposal a modification to the velocity formula, similar to the one made by Chow and Tsui [14], was made. As in [14], they aggregated a neighbor influence factor into the equation to calculate velocity, and suggested that it will help particles to escape local optima. Also, an AER (Agent-Enviroment-Rules) model is used in combination with the PSO model. This model introduced some rules for the particles, as now they are treated as agents with the ability of: memory, communication, response, cooperation and self-learning. This agents have some properties as: velocity and energy, energy is related to its fitness; they can remember their best previous positions, personally *pbest*, locally *lbest* and globally *gbest*; particles compete and cooperate between them; and have the ability to clone themselves. The last two properties made use of a few operators; a competition operator, and a clonal selection and clonal mutation operator. The technique is compared against MOPSO [18] and NSGA II [27] showing good performance.

Raquel and Naval (2005) MOPSO-CD, a proposal by Raquel and Naval [93], extends PSO to deal with MOPs by incorporating a crowding distance mechanism in the selection for global best, and deletion method of the external archive of non-dominated solutions when it becomes full. What maintains the diversity of solutions in the external archive is the crowding distance mechanism, along with a mutation operator used with a certain percentage over the entire population. The crowding distance of a solution in the external archive is an estimate of the size of the largest cuboid enclosing it without including any other point. Guides then are selected from the external archive, according to the particles that have the biggest crowding factors (least crowded areas).

Gong et al. (2005) A multi-objective PSO based on minimal particle angle was proposed by Gong et al. [46]. In this algorithm, they proposed a method where particles select their global leaders according to the minimal angle between them and the ones stored in an external archive, where non-dominated solutions are kept. The external archive, to maintain a good distribution of solutions, uses the crowding method used by NSGA II [27].

Meng et al. (2005) Meng et al. [80] proposed a co-evolutionary PSO based method to solve MOPs (CMOPSO). The method intends to locate the extreme solutions in the Pareto front for a given problem in the early stages to reduce the search time and space for the particles in the later stages. To do that m population of swarms are used to explore the extreme areas of the Pareto front (where m is the number of objectives in the problem), while another swarm searches the Pareto front. Every swarm has their own *gbest* set and they use it to find their part of the extreme Pareto front. All the $m + 1$ *gbest* sets are updated using co-evolutionary operators. Particles are updated according to the standard PSO velocity formula plus a competition mutation operator.

Reyes and Coello (2006) In [100], a study of the tuning of the parameters used by a multi-objective PSO, in particular one previously presented by them [97, 96], is presented. The importance of the study is related to the impact that the main parameters of PSO have when used for MOPs optimisation. The parameters involved in the study are the inertia weight and learning factors used in the velocity update formula, and parameters used by their own implementation. In the study three mechanisms for an on-line adaptation of the most important parameters is proposed. And their conclusion is that the design of on-line adaptation mechanisms is possible, in order to maintain and improve the quality of results.

Wang and Singh (2006) A Fuzzified Multi-Objective PSO (FMOPSO) was proposed by Wang and Singh [124], to solve a stochastic economic power dispatch which is treated as a bi-objective optimisation problem. The method used a continuously updated external archive to store the non-dominated solutions found during its execution. To maintain the external archive updated, niching and fitness sharing mechanisms are also reported as being used, although not specified. Selection of global guides is made through a "fuzzification" mechanism, where *gbest* is not seen as a point but as an area, and each solution found in that area have different possibilities of being chosen as a *gbest*.

Krami et al. (2006) Krami et al. [68] proposed an algorithm to optimize a reactive power planning, which is intended to deal with two objectives: cost of compensation and reduction of system's losses. The algorithm is mainly based on MOPSO [20], and the experimental results demonstrate its applicability.

Huang et al. (2006) Based on a Comprehensive Learning PSO (CLPSO) [74]; Huang et al. [57], proposed an adaptation of it to solve MOPs and denominated it MOCLPSO. CLPSO, uses a novel learning strategy where all particles historical best information is used to update a particles velocity. In this way, every particle learns from all the *pbests* of other particles, which in turn makes to all the particles to learn from the elite. MOCLPSO uses a non-dominated external archive to store its solutions; when exceeding its limit size, it is updated using a crowding factor similar to the one used by NSGA II [27]. Updates to *pbest*s are performed according to dominance. *gbest*s are selected randomly from the external repository.

Ho et al. (2006) Intelligent MOPSO (IMOPSO) was proposed by Ho et al. [50]. IMOPSO uses an intelligent move mechanism (IMM) and a generalized Pareto-based scale-independent fitness function (GPSISF) to solve

MOPs. The IMM is a mechanism based on orthogonal experimental design, which uses an approach to determine the next move of a particle. This in turn makes to the mechanism spend at most $2N$ objective function evaluations to find a potentially good solution[9]. For each particle IMM generates two temporary moves, one corresponding to the cognitive, and the other to its social part; this with an aim to efficiently combine good partial vectors of both parts, to generate the next particle's position. In the study a fitness function, called GPSISF, is defined and used to measure the fitness of particles in the swarm. The fitness is calculated with a tournament-like score, obtained from particles participants and based in dominance. IMOPSO uses an elite set, from where the particles select their *gbest*.

A recent proposal by Salazar Lechuga and Rowe [101, 102] will be presented in the following Chapters.

Summary

After reviewing some of the most important adaptations of PSO to handle multi-objective problems, we would like to discuss some of the main algorithmic features used by this heuristics.

One of the most important features in the modification of a PSO–to deal with MOPs, is the way in which the PSO is adjusted for the selection of leaders; and not surprisingly here is where most of the heuristics differentiate from each other. We have seen that most of them are chosen according to dominance, a niching method or density estimator, randomly, or just based

[9]Where $N \leq M$, in a M-dimensional objective vector.

on the performance on one of the objectives; the way they are finally selected is as diverse as the number of proposals we have reviewed, but generally the selection of leaders is based on these schemes.

Another characteristic, which is shared between most of them, is the use of an external archive to store their non-dominated solutions. Most of the approaches tend to use an external archive where they store non-dominated solutions which they use to guide their search. The strategy they follow to insert solutions into the archive is also as diverse as the number of heuristics that made use of an external archive.

Interestingly, except for a couple of implementations, a *gbest* model for the PSO is almost always used by all the heuristics; the *gbest* model is where all the members of the swam use their own experience *pbest* and a global best position *gbest* to update their velocities (Section 2.3.1).

2.4 Niching Methods

Usually the population of evolutionary algorithms cannot maintain diversity per se. This always results in the entire population converging to an area or a point in the search space where the best solutions found so far might lie in. This is not always convenient, particularly when we are interested to maintain diversity in the population, as is the case when solving multi-modal or multi-objective problems.

To enforce the diversity needed in the population, some methods have been developed. This methods usually are referred as *niching* methods [78]. Niching has its inspiration in ecology, where organisms have to share resources and where these resources are located in certain areas of their ecosystem, organisms will have to move accordingly to where the best location of resources are placed.

Niching methods in evolutionary algorithms try to encourage the formation of sub-populations in neighbourhoods where optimal solutions are found. Therefore, individuals that might concentrate in a small portion of the search space without the use of niching methods, will be encourage to disperse into a bigger portion of it by using them.

Originally introduced by Holland [51], one of the best well known and probably most used niching method is the fitness sharing method [103].

2.4.1 Fitness Sharing

The main idea of fitness sharing, Goldberg and Richardson [45], is to distribute a population of individuals along a set of resources. When an individual i is sharing resources with other individuals, its fitness f_i is degraded in proportion to the number and closeness to individuals that surround it. Then the fitness sharing f_i' for and individual i is defined as:

$$f_i' = \frac{f_i}{m_i} \tag{2.13}$$

where m_i is the niche count that measures the approximate amount of individuals with whom the fitness f_i will be shared, which will be the addition of a sharing function for all the members in the population

$$m_i = \sum_{j=1}^{n} sharing(d_{ij}) \tag{2.14}$$

where n is the number of individuals in the population, and d_{ij} is a measure of distance between individual i and j. The sharing function is usually a measure of similarity between the solutions being compared. The most used sharing function returns a value of 0 when the solutions are not very similar, or a value in the range $[0, 1]$ based on their similarity, a value of 1 indicates two identical solutions:

$$sharing(d_{ij}) = \begin{cases} 1 - (d_{ij}/\sigma_{share})^{\alpha} & \text{if } d_{ij} < \sigma_{share} \\ 0 & \text{Otherwise} \end{cases} \tag{2.15}$$

σ_{share} is the distance radius that we want the individuals to remain apart from each other. α is a constant that regulates the shape of the share function, this constant is usually set to 1.

The distance to measure similarity between solutions can be genotypical or phenotypical based, which one to select mainly depends on the problem to be solved. For GAs, genotypic similarity is linked to the bit-string representation of solutions, whereas the phenotypic similarity is related to their decoded values in the search space.

Deb and Goldberg [26] made further studies experimenting with genotypic and phenotypic sharing. In their study, they used Hamming distance to measure similarity in the genotype between solutions, and Euclidian distance to measure similarity in the phenotype. Their conclusions were that the use of phenotypical fitness sharing shows better results than the genotypical fitness sharing.

2.5 Dynamic Optimisation

We mentioned before that most real world problems are multi-objective, we have to add that many real world problems are also dynamic or continuously changing over a period of time. The task of solving dynamic optimisation problems is not an easy one.

As we have discussed, evolutionary methods use populations of solutions, these solutions evolve through time according to the conditions given by the problem being solved. On a static problem those conditions will keep constant through the whole evolutionary process, thus the solutions will evolve or adapt based on those stable conditions and eventually at the end of the process optimum solutions will be found. On a dynamic problem, these conditions will change at some rate. Standard evolutionary methods tend to have difficulty to adapt to environments which are continuously changing, this is due to the imminent convergence of solutions to an optimum, which turns into a decrease of diversity in the population; the lack of diversity translates in a lack of ability to adapt when a change in the environment occurs.

Evolutionary approaches have been successfully modified in order to deal with this type of problems [11, 12, 3]. Recently, PSO has also made incursion into the field of dynamic optimisation: Carlisle and Dozier [13] were one of the firsts to propose a modification to PSO to work in dynamic enviroments; Eberhart and Shi [30] successfully tracked dynamically varying parabolic functions, using a conventional PSO; Blackwell and Bentley [8] modified the PSO algorithm to better track and optimise a parabolic function where the optimum location dynamically changes, this modification is called charged PSO, and is based on the analogy of electrostatic energy with charged particles; Blackwell and Branke [7] presented a PSO variant, which was tested on a multi-modal dynamic function and compared against an evolutionary algorithm, showing promising results.

2.6 Summary

In this Chapter we have introduced different concepts which we will be using along this thesis. We have talked about multi-objective optimisation, the difficulty involved when solving this type of problems, and how many evolutionary computation techniques have been successfully solving them. We talked about particle swarm optimisation, how this technique compares with evolutionary computation techniques, and how it has also been successfully used to solve multi-objective optimisation problems. We also talked about niching methods and what they are useful for. And at the end, we talked about dynamic optimisation, its innate difficulty to be solved, and how they have started to been approached by the evolutionary computation and particle swarm optimisation communities.

Chapter 3

Particle Swarm Optimisation and Fitness Sharing to Solve Multi-Objective Problems

As we have seen in Section 2.3, Particle Swarm Optimisation (PSO) is a non-linear function optimisation technique of recent development [117]. This heuristic has good performance, low computational cost and is easy to implement. Due to those characteristics, plus the similarities that this technique share with evolutionary algorithms, evolutionary computation scientists have been attracted to study this heuristic more closely.

Originally PSO was conceived to solve single-objective problems. Some work has been done to enhance PSO to deal with multi-objective optimisation problems (see Section 2.3.2). PSO has also been paired with fitness sharing to solve multi-modal function optimisation problems [71], and previous work by Li [72] has made use of niching techniques to enhance PSO behaviour on

MOPs.

In this Chapter we will introduce a heuristic developed to solve multi-objective problems making use of particle swarm optimisation and fitness sharing, concepts we got familiar with on the previous Chapter. The heuristic will be described and then tested with different problems taken from the specialized literature, it will also be analysed and compared against other different state-of-the-art heuristics to solve multi-objective optimisation problems.

3.1 Proposed Approach

Goldberg [44], in his book, suggested the use of niche and speciation methods into the MOEA area. Back then, he thought they may be especially useful to avoid competition between distant members of the population, and in this way promoting and maintaining diversity. Following Goldberg's suggestions, multiple independent groups implemented his ideas [52]. The four main techniques are MOGA, NPGA, NSGA, and the Pareto-optimal ranking GA with sharing. This has lead us to the proposal of the work that we will describe in this section.

One of the characteristics that made PSO (or evolutionary algorithms) so attractive to solve MOPs is due to its population-based solutions mechanism. Thanks to this mechanism, these kind of heuristics are capable of providing several solutions in one execution, in contrast to traditional techniques where one execution is capable to produce just one single solution.

Our idea is to use the PSO technique to guide the search for solutions, and to use fitness sharing to spread the particles along the Pareto front to generate diversity between solutions. PSO will be the force that will drive the particles towards the Pareto front; but–evidently–PSO on its own is not capable of maintaining the diversity needed to cover a wide range of the Pareto front, it only has the strength to propel the particles towards the Pareto front and perhaps to find a small fraction of it.

Because of that reason, it is necessary to use a spread force that will promote diversity between the particles and move them apart as to cover wider areas. To accomplish that we will use fitness sharing; it will help to our algorithm to maintain diversity between solutions, and to distribute them along the Pareto front.

When solving MOPs we deal with two spaces, variable space and objective space; decision variables are mapped to objective vectors via a function evaluation. Because fitness sharing could be used in either of those two spaces we have chosen to use it under the objective space. This is based on the premises that: (a) we need to create a diversity of solutions in the objective space, right over the Pareto front (see Figure 3.1); and (b) as the study by Deb and Goldberg [26] shows a phenotypic use of fitness sharing could provide better results, this by making an analogy of seeing the variable space as the genotype and the objective space as the phenotype.

3.1.1 Algorithm

As we have seen in Section 2.3, particles need guidance from mainly two sources: (1) the best solution in the swarm (global best), and (2) their pre-

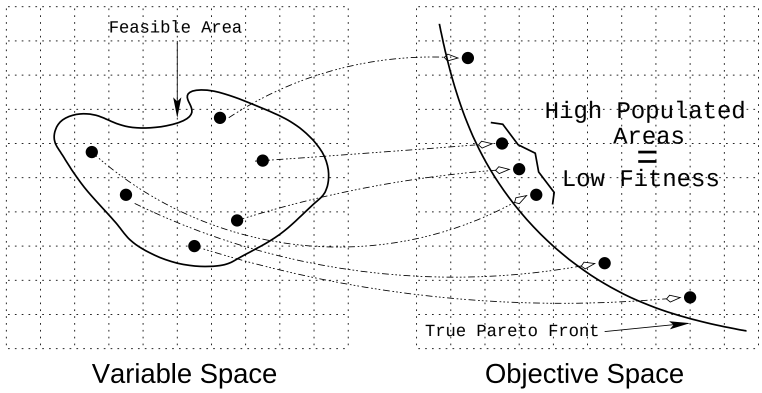


Figure 3.1: Graphical representation of solutions in the variable space mapped over the objective space. Our approach calculates fitness sharing using the location of the solutions found by the particles in the objective space.

vious best location (stored in their memory). Because inherently to multi-objective optimisation there is no single best global solution, but a set of solutions; we need to promote the use of a mechanism which enables particles to use not a single solution as a leader, but a set of solutions. One way to do this is to use a repository, as in [18], where non-dominated solutions are stored in an external archive, which contains the best solutions found so far, and that particles can use to select a leader from it to guide their search.

To maintain this repository in an updated state during each iteration of the algorithm, the best particles found–those not dominated–will be inserted in the repository (or archive), and those which are no longer non-dominated will be deleted from it. In this way, the repository will help to guide the search for the next generations and will maintain a set of not dominated solutions until the end of the run, which is what we are looking for, the set of solutions forming the Pareto front.

The flow of the algorithm is shown in Figure 3.2. An explanation follows for each of the steps given in the diagram:

1. In the first step all variables used by the algorithm are initialized. Particles ($pop[i]$) are initialized inside the search space, and their memories ($pbest[i]$) are set to the current location. The external repository ($gbest[i]$) is filled with all the non-dominated particles. And fitness sharing ($f'[i]$) is calculated for each particle in the repository.

   According to the fitness sharing principle, which in words states that particles (or solutions) which have more particles in their vicinity will be less fit than those that have fewer particles surrounding their vicinity. The fitness assigned is given by:

   $$f'[i] = c/m_i \tag{3.1}$$

   where $c = 10$; the value for c was arbitrarily chosen with the purpose to assign the same fitness value to every particle, and only being degraded according to the proximity that it has to other particles. A high value of $f'[i]$ (close to, or 10) will mean that the particle i is not surrounded by other particles, or at least that there are particles not so close to particle i; on the other hand, a small value of $f'[i]$ means that the particle i is on a crowded area and will have less oportunity to be chosen as a leader.

   $$m_i = \sum_{j=1}^{n} sharing(d_{ij}) \tag{3.2}$$

where n is the number of particles in the repository.

$$sharing(d_{ij}) = \begin{cases} 1 - (d_{ij}/\sigma_{share})^2 & \text{if } d_{ij} < \sigma_{share} \\ 0 & \text{Otherwise} \end{cases} \tag{3.3}$$

σ_{share} is the distance we want the particles to remain distant from each other; and d is a measure of distance between particles i and j.

$$d_{ij} = \sqrt{(part_i - part_j)^2} \tag{3.4}$$

2. Having assigned a fitness sharing to each particle in the repository, particles from the repository will be chosen as leaders to guide the search. They will be chosen according to a stochastic universal sampling method (Roulette Wheel) using their assigned fitness sharing values. Particles with higher levels of fitness will be selected over the less fit ones. Because fitness measurement is based on a niching method, this will drive to the particles to go into places which are less explored in the search space. The velocity for the particles is calculated, similar to Equation 2.12, as:

$$\begin{aligned} vel[i] = w \times vel[i] + c_1 \times r_1 \times (pbest[i] - pop[i]) + \\ c_2 \times r_2 \times (gbest[h] - pop[i]) \end{aligned} \tag{3.5}$$

where w is an inertia weight, c_1 and c_2 are acceleration coefficients, $vel[i]$ is the previous velocity value, r_1 and r_2 are random values between 0 and 1, $pbest[i]$ is the previous best position found by particle

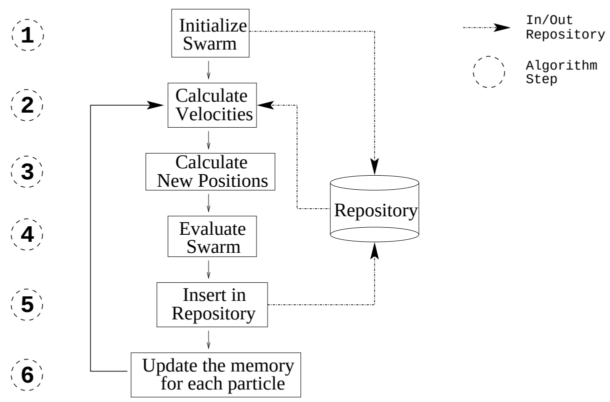


Figure 3.2: Diagram of the Algorithm

i, $gbest[h]$ is the particle h from the repository to be follow, and $pop[i]$ is the current position of the particle i in the variable space.[1]

3. New positions of the particles are calculated according to the velocities obtained in the previous step:

$$pop[i] = pop[i] + vel[i] \tag{3.6}$$

4. The new positions of the swarm are evaluated.

5. The repository is updated with the current solutions found by the particles. The criteria used to update the repository is dominance and fitness sharing, this step is detailed in the next Section.

6. Finally the memory of each particle is updated using dominance criteria; if the current location of the particle dominates the one stored in

[1]For all of our experiments, unless stated otherwise, we used an inertia weight w value of 0.4, and a value of 1.0 was given to both coefficients c_1 and c_2. Which were the values we found to work the best, after an extensive set of experiments.

its memory, the current one replaces the one in memory.

Repository

The repository is the mechanism that guides the particles towards the Pareto front, this is why is important to maintain a diverse non-dominated set of solutions in it. There are three main case that we could find when trying to insert a solution z in the repository, and could be the following: (a) when the repository is empty, (b) when the repository is not empty and not full, and (c) when the repository is full.

To deal with case (a), when the repository is empty, we just need to insert the solution z in the repository (see Figure 3.3).

In case (b), when the repository is not empty nor full, we will need to proceed to compare the solution z we want to insert against all the other solutions inside the repository. As a result of this comparison, two main situations could arise: (1) the solution z we want to insert is dominated by at least one solution in the repository, then we won't proceed to insert that solution in the repository (see Figure 3.4a); or (2) solution z is non-dominated by any of the solutions in the repository, which means that we will be inserting that solution in the repository (see Figure 3.4b and 3.4c). Also in this last case, solutions which are dominated by the solution z will be deleted from the repository, in this way we maintain the repository as the Pareto front found so far (see Figure 3.4b).

In the last case (c), when the repository is full of non-dominated solutions, we could have two main situations, just like in the previous case. With situation (1) we deal exactly the same way; when the solution z to be inserted

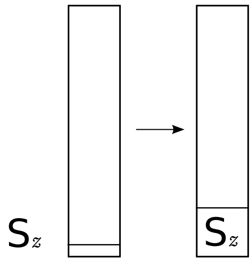
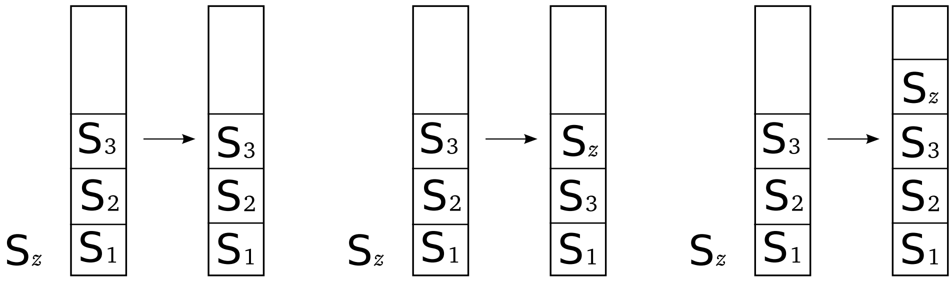


Figure 3.3: Inserting a candidate solution S_z in an empty repository.

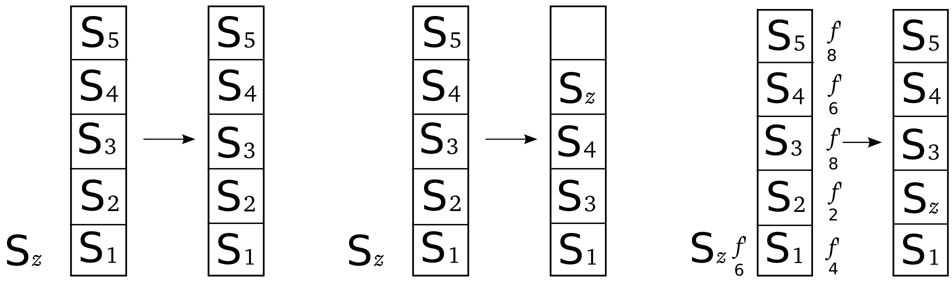


(a) If candidate solution S_z is dominated by at least one solution in the repository, then solution S_z is rejected.

(b) Solutions dominated by the candidate solution S_z are eliminated from the repository, and candidate solution S_z inserted.

(c) Candidate solution S_z and solutions in the repository are mutually non-dominated, solution S_z is inserted.

Figure 3.4: When there is at least one solution already in the repository we need to check for dominance against those solutions.



(a) If candidate solution S_z is dominated by at least one solution in the repository, then solution S_z is rejected.

(b) Solutions dominated by the candidate solution S_z are eliminated from the repository, and candidate solution S_z inserted.

(c) Fitness sharing for solution S_z is calculated, in case it is better than one of the solutions in the repository, solution z will replace it; otherwise solution z is discarded.

Figure 3.5: When the repository is full we need to check for non-dominance and fitness sharing.

is dominated by at least one solution in the repository, we ignore solution z (see Figure 3.5a). With situation (2), we deal in a slightly different way. In the previous two cases (a) and (b), we did not have a full archive and it did not matter that much if solution z dominated solutions in the repository, thus eliminating solutions from the repository; in this case this is important because if it does, it will free at least one place in the repository where solution z can fit in (see Figure 3.5b).

But in the case where solution z is non-dominated by solutions in the repository and vice-versa, there will be no place for solution z inside the repository, as all of them are non-dominated. When this situation arises we made use of fitness sharing to decide if solution z will be inserted or not; and if so, also to decide which solution will be eliminated inside the repository to cede its place to solution z. In this situation then, we will calculate the fitness sharing for the solution z that wants to get into the repository; and if is better than the worst fitness sharing for a solution in the repository, then the solution with worst fitness sharing is replaced by this new one (see Figure 3.5c). Fitness sharing for all solutions will be updated when inserting or deleting a solution from the repository. This is to maintain fitness sharing in an up to date state, just in case is used again when choosing a leader when calculating velocities, or when inserting solutions into the repository.

3.2 Measuring and Testing Performance

To assess performance in a multi-objective heuristic we need: metrics that evaluate and give a tangible value to compare against other techniques, and a set of test functions which can be used between different techniques to evaluate and compare performance between them. In this section we will talk about the metrics that we will use in our work to measure performance, and we will introduce test functions that will be used to test and compare heuristics in this work.

3.2.1 Multi-Objective Optimisation Metrics

Assessing performance of heuristics that solve multi-objective problems is a multi-objective problem per se; this is because of all of the different evaluating aspects we have to take into account when measuring the potential of a multi-objective heuristic [136]. Mainly, the objectives that a metric has to evaluate are:

- Proximity to the real Pareto front, heuristics have to give solutions as close as possible to the optimal solutions.
- Diversity among solutions, we need to have a wide range of variety in the solutions.
- Extension of the solutions found, having an area–where solutions might lie in–as large as it is possible will help to promote diversity in the solutions.

Taking these three objectives in mind, we decided to use the following metrics to measure performance:

Generational Distance [119] finds the average distance of the non-dominated set of solutions found from the Pareto optimal set:

$$GD = \frac{\sqrt{\sum_{i=1}^{n} d_i^2}}{n} \tag{3.7}$$

where d_i is the Euclidean distance between solution i from the set of n non-dominated solutions found and the closest element from the Pareto optimal set (in objective space).

Because this metric measures the average distance from the non-dominanted solutions found to the Pareto optimal set, we are interested on smaller values; as they indicate a smaller proximity to the Pareto optimal set.

Spacing [105] measures how well distributed (spaced) the solutions in the non-dominated set are:

$$S = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(d_i - \overline{d})^2} \tag{3.8}$$

where d_i is the minimum value of the sum of the absolute difference for every objective function value between the i-th solution and all the n non-dominated solutions found, $d_i = min_{j=1 \wedge j \neq i}^{n}(\sum_{m=1}^{M} |f_m^i - f_m^j|)$; $\overline{d}$ is the mean value for all d_i, $\overline{d} = \sum_{i=1}^{n} d_i/n$. Because this metric measures the standard deviations of the distances in-between the non-dominated solutions found, a smaller value indicates that these solutions are uniformly spread.

Maximum Spread [129, 130] gives a value which represents the maximum extension between the farthest solutions in the non-dominated set found; in a problem with two objectives, the value will be the Euclidean distance between the two farther solutions.

$$D = \sqrt{\sum_{m=1}^{M} (max_{i=1}^{n} f_m^i - min_{i=1}^{n} f_m^i)^2} \tag{3.9}$$

n is number of solutions in the non-dominated set, and M is the number of objectives in a given problem. In this metric a bigger value indicates better performance.

We use three more metrics, as indicators on performance, over three-dimentional functions employed to test our algorithms. These metrics were chosen based on discussions by Zitzler et al. [136] and Knowles et al. [65], which suggest the use of the following quality indicators over the previous chosen ones, specially in cases when the number of dimensions is higher. These indicators are:

Hypervolume [134, 135] This unary indicator, as a connotation to its name, measures the hypervolume of that portion of the objective space that is weakly dominated by an approximation set P'. In order to measure the hypervolume of an approximation set, a bounding reference point (at least weakly) dominated by all points should be used. In our work we are using the hypervolume difference to a reference set P, and we will refer to this indicator as I_H^- [65]. Given an approximation set P', the indicator value is defined as:

$$I_H^-(P') = I_H(P) - I_H(P') \tag{3.10}$$

in which smaller values indicate higher quality on the approximation set P'.

Epsilon [136] This indicator comprises two versions an additive and a multiplicative, and both exist in unary and binary form. In our studies the unary-additive version is the one that will be used, and we will refer to it as $I^1_{\epsilon+}$ [65]. The unary-additive epsilon indicator, is defined as:

$$I^1_{\epsilon+}(P') = I_{\epsilon+}(P', P) \tag{3.11}$$

where P is a reference set, and P' an approximation set. $I^1_{\epsilon+}(P')$ gives the minimum factor ϵ by which each point in the reference set P can be added such that the resulting transformed approximation set is weakly dominated by the approximation set P'. Thus a smaller value, means a better approximation set P'.

R There are three R indicators proposed by Hansen and Jaszkiewicz [47], which can be used to asses and compare approximation sets on the basis of a set of utility functions. An utility function u maps each point in the objective space into a measure of utility, from a set of n-dimensional objective vectors to the set of real numbers [64]. In our work, we will be using the R_3 indicator in its unary form, and we will refer to it as I^1_{R3} [65]. I^1_{R3} is defined as:

$$I^1_{R3}(P', P) = \frac{\sum_{\lambda \in \Lambda} [u^*(\lambda, P) - u^*(\lambda, P')]/u^*(\lambda, P)}{|\Lambda|} \tag{3.12}$$

where $\lambda = (\lambda_1, \ldots, \lambda_n) \in \Lambda$ stands for a particular weight vector, u^* is the maximum value reached by the utility function u_λ with weight vector λ on a reference set P, or an approximation set P', i.e. $u^*(\lambda, P) = max_{z \in P}\ u_\lambda(z)$. The parameterized utility function u_λ used is the augmented Tchebycheff function,

$$u_\lambda(z) = -\left(\max_{j \in 1 \ldots n} \lambda_j |z_j^* - z_j| + \rho \sum_{j \in 1 \ldots n} |z_j^* - z_j| \right) \tag{3.13}$$

where ρ is a small positive real number. According to Hansen and Jaszkiewicz [47], this indicator follows an approach often used in single objective optimization, where an approximate solution is evaluated by the radio of its value to that of a fixed bound, for instance the optimal value. As such, lower values are better, with $(-)$ infinity being best, and higher values are worse, with $(+)$ infinity being worst.

For the last three metrics, we have to say, that all three of them are Pareto compliant indicators; and when two Pareto compliant indicators contradict each other, when comparing two approximation sets, then it will be implied that the two sets are incomparable. Also, in our experiments, the implementations to calculate the values for the last three indicators, were based on those found on the PISA implementation [9, 65].

3.2.2 Test Functions

To test the performance of multi-objective optimisation approaches, there are several test functions whose Pareto optimal solutions sets are known. Many of this problems can be found in the specialised literature [118, 23, 24, 28, 29, 21, 129, 130]. Here we will list the ones we have used to test our

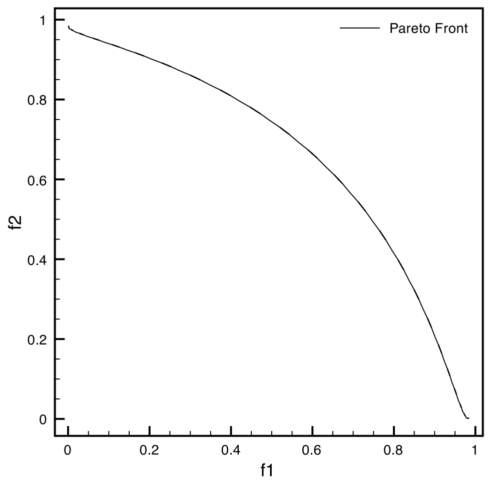


Figure 3.6: Pareto optimal front in the objective space for Fonseca's test problem.

heuristic.

Test function 1

The first test function, proposed by Fonseca and Fleming [43], is a bi-objective optimisation problem:

$$\text{FON}: \begin{cases} \text{Minimize} & f_1(\vec{x}) = 1 - \exp\left(-\sum_{i=1}^{n}\left(x_i - \frac{1}{\sqrt{n}}\right)^2\right) \\ \text{Minimize} & f_2(\vec{x}) = 1 - \exp\left(-\sum_{i=1}^{n}\left(x_i + \frac{1}{\sqrt{n}}\right)^2\right) \\ \text{where:} & -4 \leq x_i \leq 4 \qquad i = 1, 2, 3. \end{cases} \tag{3.14}$$

The non-convex Pareto optimal front for this problem can be seen in Figure 3.6.

Test function 2

Our second test function, is a maximisation problem proposed by Poloni [24, pg. 328]:

$$\text{POL}: \begin{cases} \text{Maximize} & f_1(x_1, x_2) = -[1 + (A_1 - B_1)^2 + (A_2 - B_2)^2], \\ \text{Maximize} & f_2(x_1, x_2) = -[(x_1 + 3)^2 + (x_2 + 1)^2], \\ \text{where:} & A_1 = 0.5 \sin 1 - 2 \cos 1 + \sin 2 - 1.5 \cos 2, \\ & A_2 = 1.5 \sin 1 - \cos 1 + 2 \sin 2 - 0.5 \cos 2, \\ & B_1 = 0.5 \sin x_1 - 2 \cos x_1 + \sin x_2 - 1.5 \cos x_2, \\ & B_2 = 1.5 \sin x_1 - \cos x_1 + 2.0 \sin x_2 - 0.5 \cos x_2, \\ & -\pi \leq (x_1, x_2) \leq \pi. \end{cases} \tag{3.15}$$

This problem has a non-convex disconnected Pareto front as shown in Figure 3.7.

Test function 3

Kursawe [70] with two-objectives to optimise is our third test function:

$$\text{KUR}: \begin{cases} \text{Minimize} & f_1(\vec{x}) = \sum_{i=1}^{n-1} \left(-10 e^{(-0.2)*\sqrt{x_i^2 + x_{i+1}^2}}\right) \\ \text{Minimize} & f_2(\vec{x}) = \sum_{i=1}^{n} \left(|x_i|^{0.8} + 5 \sin (x_i)^3\right) \\ \text{where:} & -5 \leq x_i \leq 5 \qquad i = 1, 2, 3. \end{cases} \tag{3.16}$$

This problem has three non-convex disconnected Pareto optimal fronts, this can be appreciated in Figure 3.8.

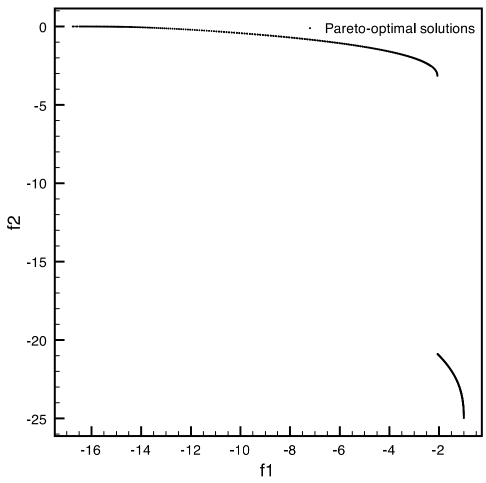


Figure 3.7: Pareto optimal solutions in the objective space for Poloni's test problem.

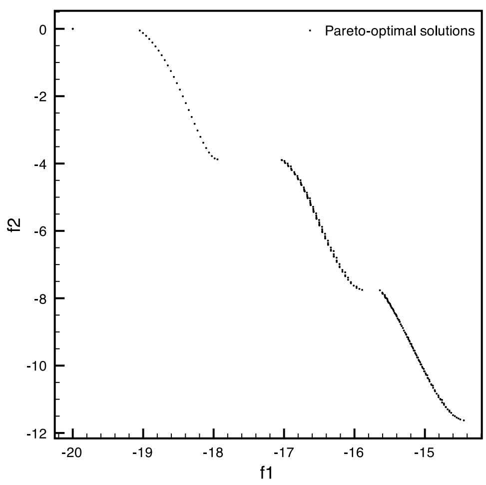


Figure 3.8: Pareto optimal solutions in the objective space for Kursawe's test problem.

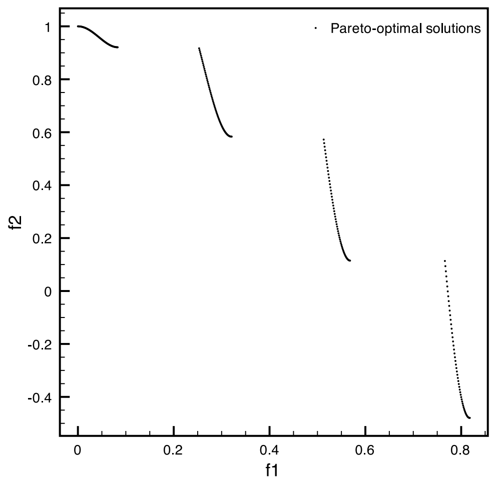


Figure 3.9: Pareto optimal solutions in the objective space for Deb's test problem.

Test function 4

Our fourth test function, proposed by Deb [23], is:

$$\text{DEB}: \begin{cases} \text{Minimize} & f_1(x_1, x_2) = x \\ \text{Minimize} & f_2(x_1, x_2) = (1 + 10y) * \left[1 - \left(\frac{x}{1+10y}\right)^{\alpha} - \frac{x}{1+10y}\sin(2\pi q x)\right] \\ \text{where:} & 0 \leq (x_1, x_2) \leq 1 \quad q = 4 \quad \alpha = 2. \end{cases} \tag{3.17}$$

Deb's problem has four non-convex disconnected Pareto optimal front sets, see Figure 3.9.

Test function 5

This, and the following two functions are from the scalable test problems proposed by Deb et al. [28, 29], the first function used in this work is the

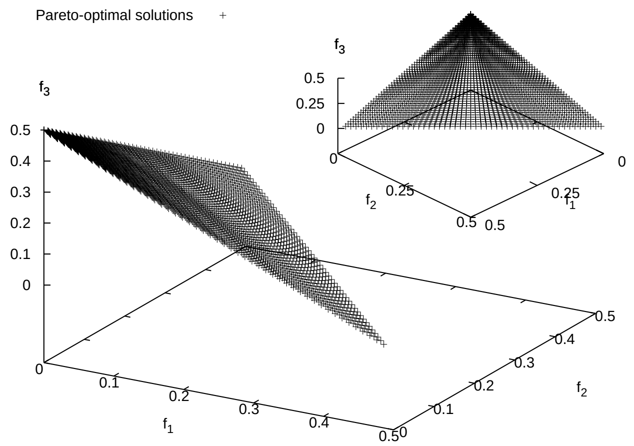


Figure 3.10: Pareto optimal solutions in the objective space for DTLZ1 test problem.

one referred as DTLZ1, and has three functions to minimize:

$$\text{DTLZ1}: \begin{cases} \text{Minimize} & f_1(\vec{x}) = \frac{1}{2}x_1x_2(1+g(\vec{x})) \\ \text{Minimize} & f_2(\vec{x}) = \frac{1}{2}x_1(1-x_2)(1+g(\vec{x})) \\ \text{Minimize} & f_3(\vec{x}) = \frac{1}{2}(1-x_1)(1+g(\vec{x})) \\ \text{where:} & g(\vec{x}) = 100\Big[|\vec{x}_m| + \sum\limits_{i=n-|\vec{x}_m|}^{n} ((x_i-0.5)^2 \\ & \qquad\qquad - \cos(20\pi(x_i-0.5)))\Big] \\ \text{subject to:} & 0 \le x_i \le 1, \quad i=1,2,\ldots,n, \quad n=7, \quad |\vec{x}_m| = 5. \end{cases} \tag{3.18}$$

The difficulty in this problem is to converge to the hyper-plane, as the search space contains several local Pareto-optimal fronts, see Figure 3.10.

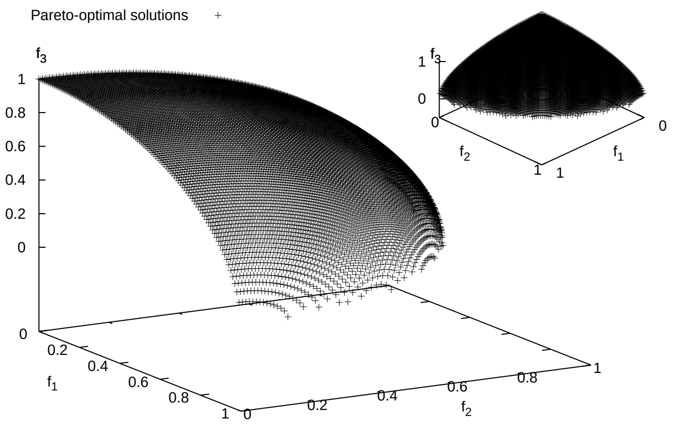


Figure 3.11: Pareto optimal solutions in the objective space for DTLZ2 test problem.

Test function 6

The second test function used from Deb et al. [28, 29] is:

$$
\text{DTLZ2}: \begin{cases} \text{Minimize} & f_1(\vec{x}) = (1+g(\vec{x}))\cos(x_1\pi/2)\cos(x_2\pi/2) \\ \text{Minimize} & f_2(\vec{x}) = (1+g(\vec{x}))\cos(x_1\pi/2)\sin(x_2\pi/2) \\ \text{Minimize} & f_3(\vec{x}) = (1+g(\vec{x}))\sin(x_1\pi/2) \\ \text{where:} & g(\vec{x}) = \sum\limits_{i=n-|\vec{x}_m|}^{n} (x_i - 0.5)^2 \\ \text{subject to:} & 0 \leq x_i \leq 1, \quad i = 1,2,\ldots,n, \quad n = 12, \quad |\vec{x}_m| = 10. \end{cases} \tag{3.19}
$$

According to its authors, this function can be used to investigate an algorithm's ability to scale up its performance in large number of objectives, see Figure 3.11.

Test function 7

The last problem chosen to test our algorithms, is DTLZ7, also found in [28, 29]. This problem, like the previous two, has three functions to be minimized:

$$\text{DTLZ7}: \begin{cases} \text{Minimize} & f_1(\vec{x}) = x_1 \\ \text{Minimize} & f_2(\vec{x}) = x_2 \\ \text{Minimize} & f_3(\vec{x}) = (1+g(\vec{x}))h(f_1, f_2, g) \\ \text{where:} & g(\vec{x}) = 1 + \frac{9}{|\vec{x}_m|} \sum\limits_{i=n-|\vec{x}_m|}^{n} x_i \\ & h(f_1, f_2, g) = M - \sum\limits_{i=1}^{M-1} \left[\frac{f_i}{1+g}(1+\sin(3\pi f_i)) \right] \\ \text{subject to:} & 0 \leq x_i \leq 1, \quad i = 1, 2, \ldots, n, \quad n = 22, \quad |\vec{x}_m| = 20. \end{cases} \tag{3.20}$$

This problem can test the ability of an algorithm to maintain sub-populations in different Pareto-optimal regions, see Figure 3.12.

3.3 Test and Comparison

We have compared our approach (which we will call MOPSO-fs) against three other well known techniques in the multi-objective literature. The techniques are: MOPSO [20], NSGA-II [25] and PAES [62][2]; which we have described in Section 2.2.2.

For the first four test functions (test function 1, 2, 3 and 4), we have

[2]Source code for these heuristics was written by their respective authors and obtained from [16].

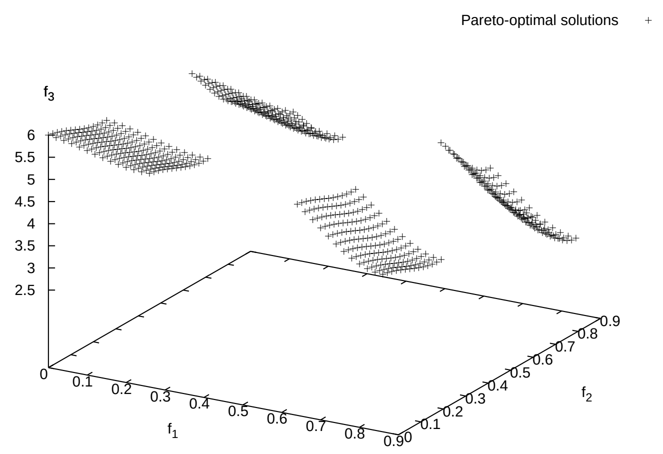


Figure 3.12: Pareto optimal solutions in the objective space for DTLZ7 test problem.

performed two sets of experiments. In the first set of experiments we have set the heuristics to find only 10 non-dominated solutions per run, and in the second set 100 non-dominated solutions. The purpose of performing this two sets of experiments, is to show how well our technique is making use of fitness sharing. Our believe is that, using a small size in the repository, our technique will still spread its solutions thanks to the use of fitness sharing.

To allow a fair comparison between all the heuristics, we performed the same number of evaluations to the objective function in each test function. To compare and obtain statistics for each test function, we performed 30 runs for each technique used. The parameters used by MOPSO, NSGA-II and PAES for all the test functions were the following:

MOPSO used a mutation rate of 0.05, 30 divisions for its adaptive grid, and a real number representation, for the first set of experiments a population of 10 particles and a repository size of 10 particles, for the second set of experiments a population of 100 particles and repository size of 100 particles.

NSGA-II used a crossover rate of 0.8, a mutation probability of $1/x$, where x is the number of variables for the given problem, and a real number representation with tournament selection, for the first set of experiments a population of 10 individuals and for the second set of experiments a population of 100 individuals.

PAES used a depth value of 5, a mutation probability of $1/L$, where L refers to the length of the chromosomic binary string, that encodes the decision variables, for the first set of experiments an archive size of 10 individuals and for the second an archive size of 100.

MOPSO-fs used a repository size of 10 particles for one set of experiments, for when a population of 10 particles was used, and a repository size of 100 particles was used, for when a population of 100 particles was used, a σ_{share} value was empirically set for each test function. The value of σ_{share} will be specified in each test function section.

Values for both MOPSOs, in the calculation of the velocity formula, were of 0.4 for the w, and 1.0 for c_1 and c_2 coefficients.

The number of evaluations for each of the test functions will be stated in each test function section. In the following all the test functions will be minimisation problems, unless stated otherwise.

The runs we choose to plot, were selected based on the median given by one of the metrics or indicators used; i.e., out of the 30 runs for each heuristic, the one closer to the median of the GD metric, or the I_H^- indicator, was the one plotted for that particular test function.

3.3.1 Test function 1

The first test function used is the one proposed by Fonseca [43] (equation 3.14, here named FON).

For this problem all the heuristics were set to evaluate the objective function $30,000$ times (in both set of experiments, when trying to find 10 and 100 non-dominated solutions). Our technique was set with a σ_{share} value of 0.1 and 0.01 for the first and second set of experiments, respectively. In table 3.1 we can observe the statistical results obtained when comparing the four different approaches. In Figure 3.13 we can see a graphical comparison of the results for the four different techniques.

3.3.2 Test function 2

Our second test function, is a maximisation problem proposed by Poloni [24, pg. 328] (named POL, see equation 3.15). For this test function, our technique used a σ_{share} value of 2.0 and 0.2 for the first and second set of experiments, respectively. All the heuristics performed $10,000$ evaluations to the objective function. Table 3.2 has the statistical values that we obtained from measuring the results, and Figure 3.14 has the graphical representations of the set of non-dominated solutions found by each of the heuristics.

3.3.3 Test function 3

Kursawe's [70] (KUR, see equation 3.16) is our third test function. For this test function a σ_{share} value of 1.0 and 0.1 was used for the first and second set of experiment, respectively, and the number of evaluations for the test

10 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.0044554	0.0012926	0.012297	0.014908
	S.D.	0.002052	0.00048675	0.0044599	0.0080007
	Med.	0.0040799	0.001341	0.011456	0.015648
	t-test	—	2.6841E-11 ∗	3.4461E-12 ∗	3.8221E-09 ∗
S	*Avg.*	0.034625	0.0091471	0.099403	0.088754
	S.D.	0.013174	0.0063679	0.029311	0.11411
	Med.	0.033343	0.0093512	0.10001	0.029709
	t-test	—	1.7476E-13 ∗	6.9662E-16 ∗	0.012401 ∗
D	*Avg.*	1.2604	0.15049	1.3883	0.72016
	S.D.	0.056672	0.082446	9.8261E-06	0.11346
	Med.	1.253	0.17435	1.3883	0.68459
	t-test	—	3.2629E-54 ∗	6.7038E-18 ∗	3.6721E-31 ∗
100 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.00077636	0.00075552	0.00079511	0.0043795
	S.D.	2.5024E-05	2.361E-05	3.1602E-05	0.0016816
	Med.	0.00077491	0.00075581	0.00079504	0.004659
	t-test	—	0.0015722 ∗	0.013491 ∗	5.9544E-17 ∗
S	*Avg.*	0.0037019	0.0088552	0.0077617	0.0088659
	S.D.	0.00049929	0.00074796	0.00058568	0.015363
	Med.	0.0036033	0.0088096	0.0078184	0.0030425
	t-test	—	4.3459E-38 ∗	3.9799E-36 ∗	0.070872
D	*Avg.*	1.3726	1.3645	1.3883	0.73446
	S.D.	0.0079205	0.010725	0.00014836	0.10007
	Med.	1.3723	1.3657	1.3882	0.71411
	t-test	—	0.0014513 ∗	1.4739E-15 ∗	1.4213E-40 ∗

Table 3.1: This table shows statistical results (average, standard deviation, median) of the metrics used over test function 1 (FON). It also shows results of a Student's t-test study, when comparing MOPSO-*fs* against the other three heuristics. [∗] Means a significant difference between results, with at least a 95% confidence level.

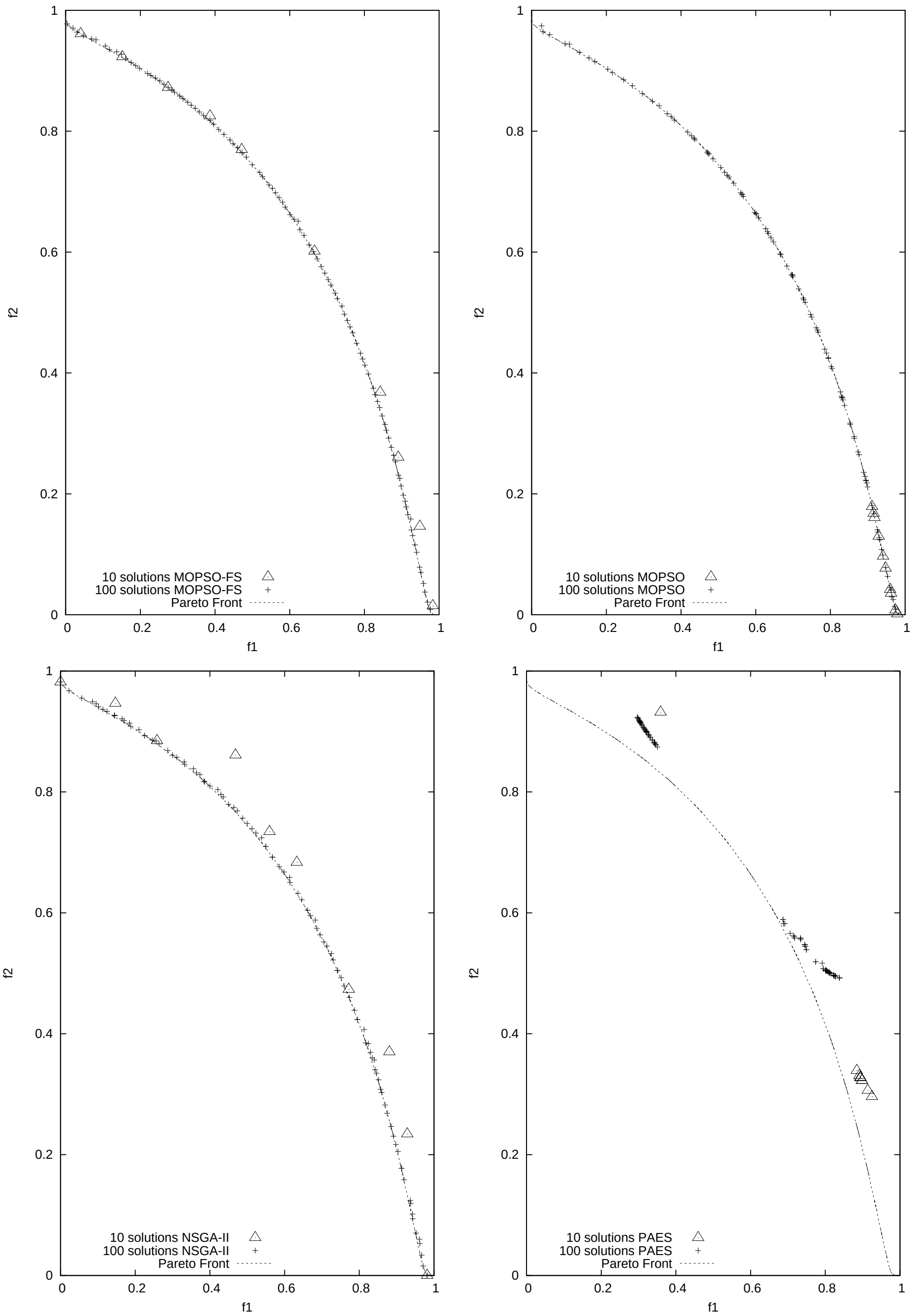


Figure 3.13: This graphical results were obtained for test function 1 (FON).

10 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.030971	0.04759	0.18864	0.08499
	S.D.	0.064904	0.083993	0.19684	0.17622
	Med.	0.0068782	0.0052603	0.048109	0.020126
	t-test	—	0.39467	0.00010421 ∗	0.12057
S	*Avg.*	0.61764	0.40905	1.6822	3.0543
	S.D.	0.15346	0.55845	0.48183	3.11
	Med.	0.59842	0.3444	1.6729	1.5117
	t-test	—	0.053312	1.2177E-16 ∗	6.9467E-05 ∗
D	*Avg.*	27.913	8.4825	30.753	20.177
	S.D.	4.4495	9.4488	1.3891	8.3684
	Med.	29.129	4.4387	29.561	22.935
	t-test	—	1.5365E-14 ∗	0.0014837 ∗	3.6787E-05 ∗
100 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.0089913	0.011953	0.00347	0.021437
	S.D.	0.013717	0.015173	0.0075011	0.037027
	Med.	0.0026618	0.0025485	0.0020762	0.0032783
	t-test	—	0.43102	0.057947	0.089608
S	*Avg.*	0.1048	0.14831	0.09264	0.20094
	S.D.	0.093952	0.090368	0.0075393	0.22917
	Med.	0.061205	0.10632	0.093508	0.13357
	t-test	—	0.072643	0.48268	0.037774 ∗
D	*Avg.*	30.105	30.268	29.639	23.744
	S.D.	1.13	1.2454	0.47579	8.4057
	Med.	29.611	29.526	29.567	27.808
	t-test	—	0.59567	0.041832 ∗	0.00012715 ∗

Table 3.2: This table shows statistical results (average, standard deviation, median) of the metrics used over test function 2 (POL). It also shows results of a Student's *t*-test study, when comparing MOPSO-*fs* against the other three heuristics. [∗] Means a significant difference between results, with at least a 95% confidence level.

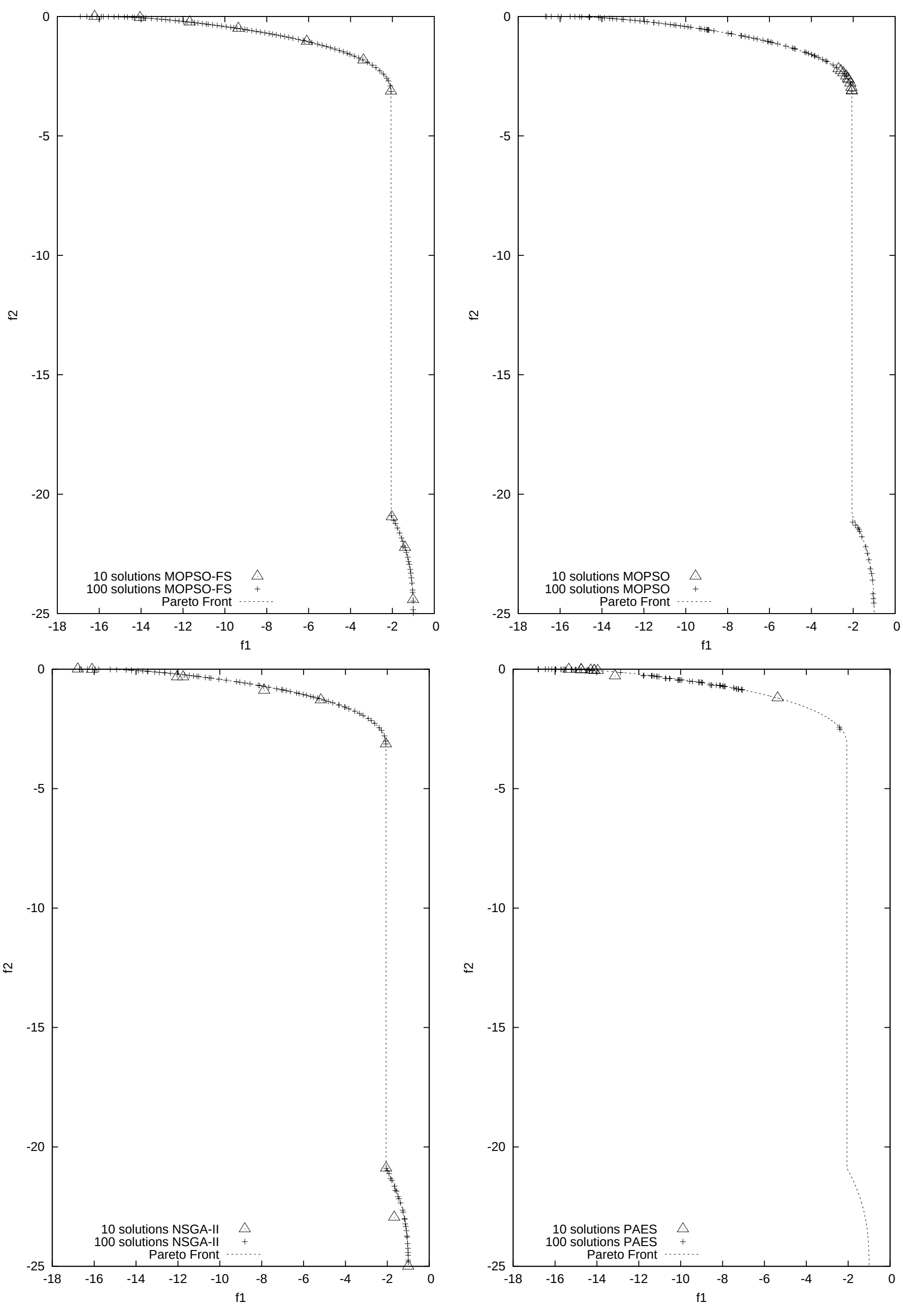


Figure 3.14: This plots correspond to the results obtained for test function 2 (POL).

function was $30,000$ times. Table 3.3 contains the statics of the comparison for this test function between the 4 heuristics. Figure 3.15 has a graphical representation of the set of non-dominated solutions for all of the techniques.

3.3.4 Test function 4

Our fourth test function was proposed by Deb [23] (DEB, see equation 3.17). For this test function we used a σ_{share} value of 0.2 and 0.01 for the first and second set of experiments, respectively, and each of the heuristics performed $5,000$ evaluations to the objective function. Table 3.4 and Figure 3.16 correspond to statistics and graphical representations to this problem, respectively.

3.3.5 Test function 5

For our fifth test function proposed by Deb et al. [28, 29], called DTLZ1 (see equation 3.18), we used a σ_{share} value of 0.1 for our set of experiments, and each of the heuristics performed $4,000$ evaluations to the objective function. Table 3.5 and Figure 3.17 correspond to statistics and graphical representations to this problem, respectively. For this problem, the indicators I_H^- and I_{R3}^1 used the vector $(25, 11, 19)$ as a *nadir* point; the indicator I_{R3}^1, for ρ used a value of 0.01, for s a value of 30, and as an ideal vector $(0, 0, 0)$. The reference set used in here can be found in [16].

		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
10 Non-Dominated Solutions					
GD	*Avg.*	0.032708	0.0048934	0.063918	0.043096
	S.D.	0.026223	0.0015383	0.044372	0.069988
	Med.	0.027673	0.0051703	0.060806	0.011842
	t-test	—	2.9083E-07 ∗	0.0015764 ∗	0.44958
S	*Avg.*	0.31937	0.030711	0.3097	1.1229
	S.D.	0.10026	0.041703	0.25404	0.52452
	Med.	0.29345	0.011041	0.23538	1.163
	t-test	—	4.9865E-21 ∗	0.84692	2.4125E-11 ∗
D	*Avg.*	12.079	0.51893	2.9609	9.6193
	S.D.	0.76027	0.65234	1.2822	2.579
	Med.	12.279	0.20362	2.7998	9.143
	t-test	—	3.4271E-55 ∗	1.1947E-39 ∗	5.3862E-06 ∗
100 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.004823	0.0042627	0.0042541	0.0058383
	S.D.	0.00080216	0.00066504	0.0021478	0.0076039
	Med.	0.0047524	0.0040873	0.0035881	0.0037764
	t-test	—	0.0046414 ∗	0.17939	0.46996
S	*Avg.*	0.078458	0.095401	0.06722	0.20549
	S.D.	0.015005	0.019198	0.029682	0.098219
	Med.	0.079328	0.10116	0.054967	0.19038
	t-test	—	0.00033964 ∗	0.069317	2.8995E-09 ∗
D	*Avg.*	12.931	12.906	11.087	11.988
	S.D.	0.029387	0.039191	0.4694	1.2002
	Med.	12.935	12.918	11.175	12.401
	t-test	—	0.0069753 ∗	2.784E-29 ∗	6.5934E-05 ∗

Table 3.3: This table shows statistical results (average, standard deviation, median) of the metrics used over test function 3 (KUR). It also shows results of a Student's t-test study, when comparing MOPSO-*fs* against the other three heuristics. [∗] Means a significant difference between results, with at least a 95% confidence level.

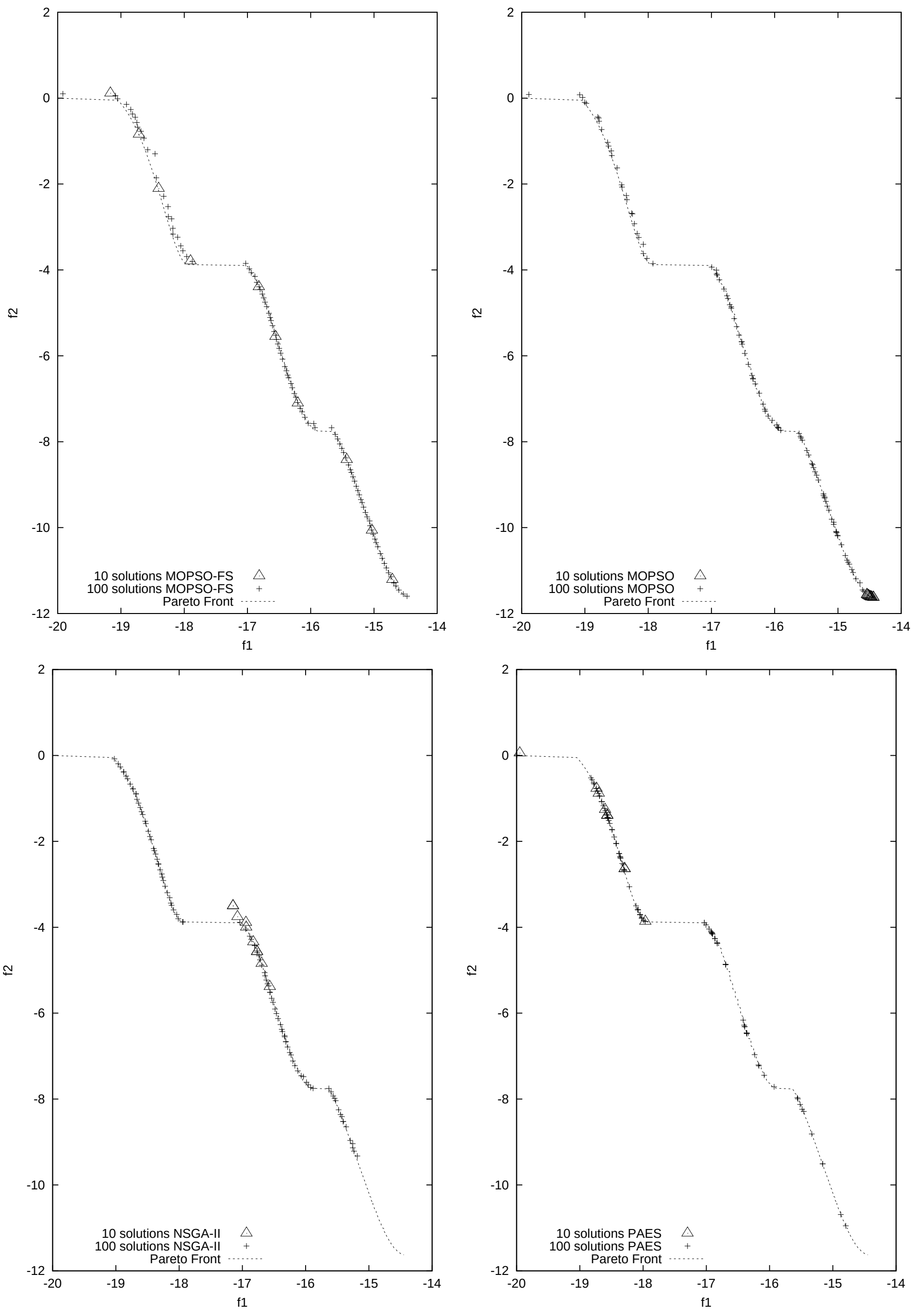


Figure 3.15: This plots correspond to the results obtained for test function 3 (KUR).

10 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.0060346	0.0088847	0.0031543	0.10488
	S.D.	0.024673	0.014328	0.0056661	0.29743
	Med.	0.001369	0.00074505	0.00089165	0.0029515
	t-test	—	0.58638	0.53561	0.074848
S	*Avg.*	0.060185	0.060908	0.11928	0.23787
	S.D.	0.031568	0.18476	0.041631	0.39435
	Med.	0.058959	0.015217	0.12205	0.16223
	t-test	—	0.98321	6.4912E-08 ∗	0.016891 ∗
D	*Avg.*	1.5812	0.38268	1.5814	2.356
	S.D.	0.45542	0.42851	0.41456	2.2331
	Med.	1.6515	0.22195	1.6903	1.6906
	t-test	—	4.9781E-15 ∗	0.9983	0.067655
100 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.001967	0.00030703	0.00032999	0.011497
	S.D.	0.0043378	2.6133E-05	2.4922E-05	0.023663
	Med.	0.00034952	0.00030767	0.00032126	0.00030953
	t-test	—	0.040461 ∗	0.043211 ∗	0.034136 ∗
S	*Avg.*	0.016964	0.0088065	0.0073167	0.03591
	S.D.	0.032405	0.0021084	0.00059085	0.047531
	Med.	0.0047589	0.0083776	0.0073142	0.016216
	t-test	—	0.17412	0.10843	0.076446
D	*Avg.*	1.8394	1.687	1.6903	2.4842
	S.D.	0.38098	0.0057372	9.2083E-05	1.572
	Med.	1.6922	1.6896	1.6903	1.6934
	t-test	—	0.032479 ∗	0.036204 ∗	0.033053 ∗

Table 3.4: This table shows statistical results (average, standard deviation, median) of the metrics used over test function 4 (DEB). It also shows results of a Student's *t*-test study, when comparing MOPSO-*fs* against the other three heuristics. [∗] Means a significant difference between results, with at least a 95% confidence level.

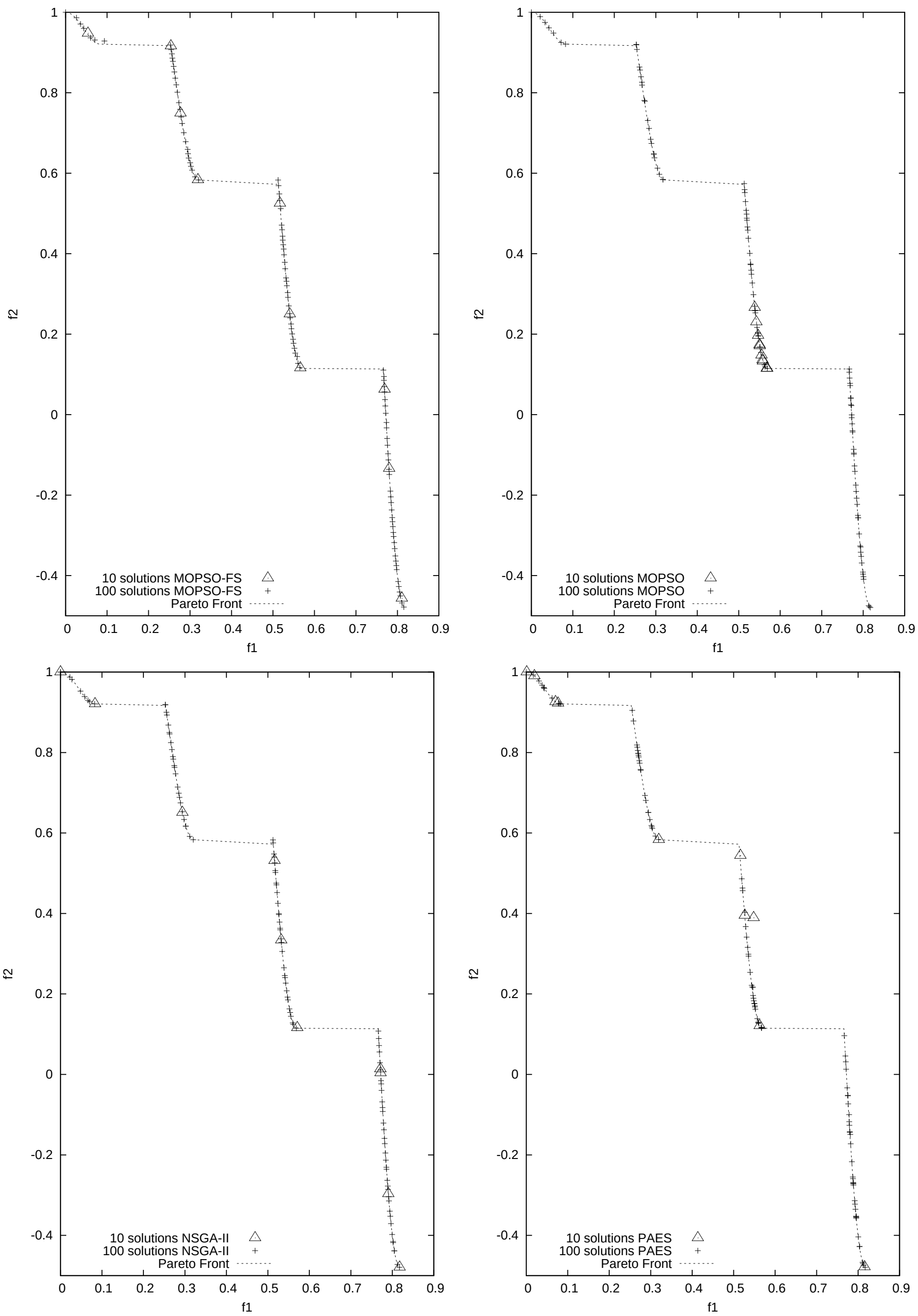


Figure 3.16: This plots correspond to the results obtained for test function 4 (DEB).

100 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
I_H^-	*Avg.*	112.35089	8.92376	-2.80144	9.29146
	S.D.	247.59312	15.20915	0.00073	10.03171
	Med.	16.08574	0.27520	-2.80163	6.47992
	t-test	—	0.02607 $*$	0.01353 $*$	0.02643 $*$
$I_{\epsilon+}^1$	*Avg.*	1.53371	0.93732	0.05484	1.21368
	S.D.	0.97677	0.53416	0.00708	0.39892
	Med.	1.36349	0.79785	0.05326	1.27249
	t-test	—	0.00478 $*$	0.00000 $*$	0.10205
I_{R3}^1	*Avg.*	0.01623	0.00783	0.00009	0.00931
	S.D.	0.01407	0.00480	0.00001	0.00334
	Med.	0.01238	0.00625	0.00009	0.00918
	t-test	—	0.00303 $*$	0.00000 $*$	0.01123 $*$

Table 3.5: This table shows statistical results (average, standard deviation, median) of the metrics used over test function 5 (DTLZ1). It also shows results of a Student's t-test study, when comparing MOPSO-*fs* against the other three heuristics. [$*$] Means a significant difference between results, with at least a 95% confidence level.

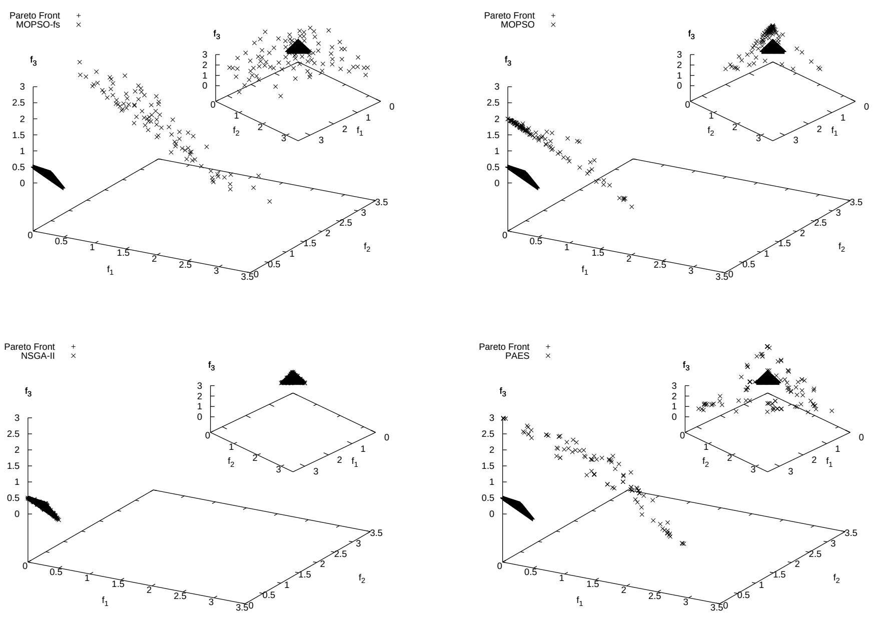


Figure 3.17: This plots correspond to the results obtained for test function 5 (DTLZ1).

100 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
I_H^-	*Avg.*	0.05766	1.37978	0.07636	0.13089
	S.D.	0.00853	0.06415	0.00749	0.05157
	Med.	0.05532	1.37905	0.07659	0.12001
	t-test	—	0.00000 ∗	0.00000 ∗	0.00000 ∗
$I_{\epsilon+}^1$	*Avg.*	0.11182	0.77761	0.12522	0.20013
	S.D.	0.02246	0.03348	0.01924	0.07008
	Med.	0.10798	0.77760	0.12663	0.17024
	t-test	—	0.00000 ∗	0.01602 ∗	0.00000 ∗
I_{R3}^1	*Avg.*	0.00003	0.16698	0.00042	0.00534
	S.D.	0.00096	0.00907	0.00054	0.00441
	Med.	-0.00017	0.16736	0.00026	0.00460
	t-test	—	0.00000 ∗	0.05697	0.00000 ∗

Table 3.6: This table shows statistical results (average, standard deviation, median) of the metrics used over test function 6 (DTLZ2). It also shows results of a Student's t-test study, when comparing MOPSO-*fs* against the other three heuristics. [∗] Means a significant difference between results, with at least a 95% confidence level.

3.3.6 Test function 6

This test function, proposed by Deb et al. [28, 29], is named DTLZ2 (see equation 3.19). For this test function, our technique used a σ_{share} value of 0.1 for this set of experiments. All the heuristics performed 500 evaluations to the objective function. Table 3.6 has the statistical values that we obtained from measuring the results, and Figure 3.18 has the graphical representations of the set of non-dominated solutions found by each of the heuristics. The values for the *nadir* vector for indicators I_H^- and I_{R3}^1 are $(1.2, 1.4, 1.7)$, the I_{R3}^1 indicator used a ρ of 0.01, a s value of 30, and an ideal vector of $(0, 0, 0)$.

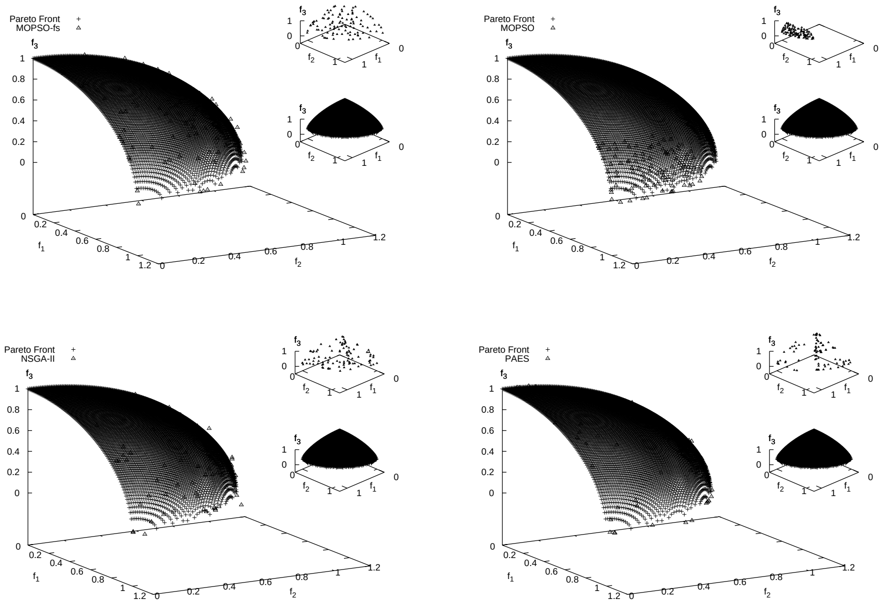


Figure 3.18: This plots correspond to the results obtained for test function 6 (DTLZ2).

100 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
I_H^-	*Avg.*	2.01224	4.87124	0.23785	0.51531
	S.D.	0.51274	1.16930	0.18460	0.55115
	Med.	1.95138	4.94397	0.14067	0.30596
	t-test	—	0.00000 ∗	0.00000 ∗	0.00000 ∗
$I_{\epsilon+}^1$	*Avg.*	2.86735	3.42265	0.57183	0.39104
	S.D.	0.77728	0.69042	0.66072	0.21360
	Med.	2.86738	3.32486	0.21493	0.31017
	t-test	—	0.00490 ∗	0.00000 ∗	0.00000 ∗
I_{R3}^1	*Avg.*	0.07823	0.22366	0.01016	0.01096
	S.D.	0.02225	0.07275	0.01386	0.01164
	Med.	0.07564	0.20795	0.00206	0.00610
	t-test	—	0.00000 ∗	0.00000 ∗	0.00000 ∗

Table 3.7: This table shows statistical results (average, standard deviation, median) of the metrics used over test function 7 (DTLZ7). It also shows results of a Student's t-test study, when comparing MOPSO-*fs* against the other three heuristics. [∗] Means a significant difference between results, with at least a 95% confidence level.

3.3.7 Test function 7

Deb's et al. DTLZ7 [28, 29] (see equation 3.20) is our last test function. For this test function a σ_{share} value of 0.05 was used for this set of experiments, and the number of evaluations for this test function was $1,000$ times. Table 3.7 contains the statics of the comparison for this test function between the 4 heuristics. Figure 3.19 has a graphical representation of the set of non-dominated solutions for all of the techniques. For this function the *nadir* reference point for indicators I_H^- and I_{R3}^1 $(1, 1, 11)$ was used; indicator I_{R3}^1 used a ρ value of 0.01, an s of 30, and the ideal vector $(0, 0, 0)$.

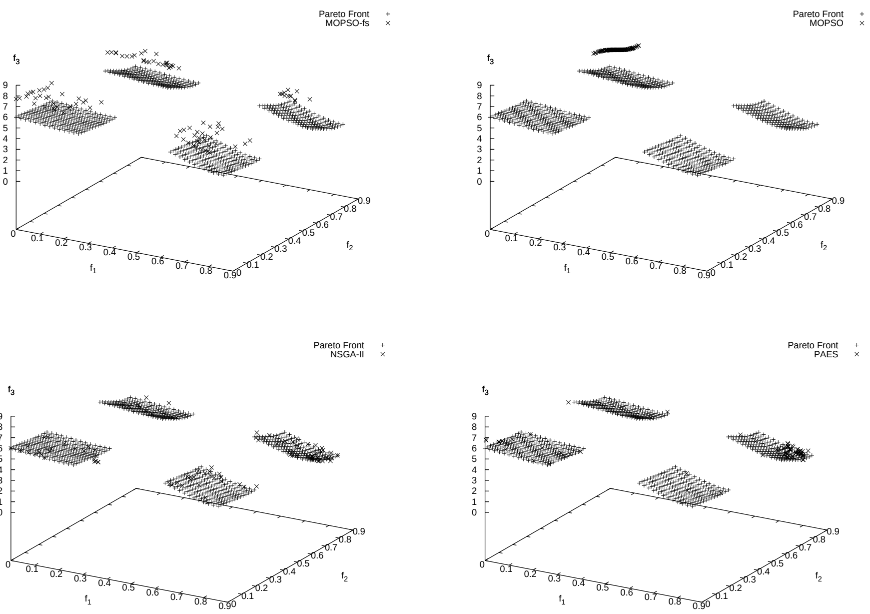


Figure 3.19: This plots correspond to the results obtained for test function 7 (DTLZ7).

3.4 Discussion of Results

Regarding our first set of experiments; which involved finding 10 non-dominated solutions, in two-objective test functions (test function 1, 2, 3 and 4). We can appreciate that our implementation obtains better results than the other techniques, from the perspective of a graphical representation of results (see Figures 3.13, 3.14, 3.15 and 3.16). All the solutions found by MOPSO-*fs*, are evenly distributed. We attribute this to fitness sharing, which uniformly distributes the particles along the Pareto front, regardless of the small number of solutions.

For the first test function, from a statistical point of view, as we can observe in Table 3.1, MOPSO has the better values for the GD and S metric, but this is due to the fact that MOPSO is concentrating all its solutions in a very small portion of the Pareto front (see Figure 3.13), and not necessarily due to a better distributed set of solutions. NSGA-II is the only heuristic that graphically (Figure 3.13) shows comparable results to MOPSO-*fs*, but statistically the technique here developed–MOPSO-*fs*–shows better results to approach the Pareto front and also to disperse the solutions when performing both sets of experiments, when finding a set of 10 and 100 solutions (smaller GD and S values, Table 3.1). In order to claim that the statistical results of our algorithm are significantly better than the others, we performed a Student's *t-test*. See Table 3.1, when there is an [$*$] in the *t-test* pair comparison, it means that a significant difference between the two sets of experiments exist, with at least a 95% of confidence level. We can see that in all cases (except for one), there is a significant statistical difference between the pair of results being compared.

For the second test function, a very similar situation as the one presented with the first test function can be observed. MOPSO presents better values for the S metric, when finding 10 non-dominated solutions (see Table 3.2), but this is due to a concentration of all its solutions in a very small portion of the Pareto front (Figure 3.14); to confirm it, we can see that metric D is very small for MOPSO, whereas MOPSO-*fs* has a much bigger value, and this has a significant statistical difference of at least a 95% of confidence level on the Student's *t-test*. NSGA-II again is the only heuristic with comparable graphical results to our technique, but again MOPSO-*fs* presents better values for GD and S (Table 3.2); which show a significant statistical difference between results, with at least a 95% confidence level. When finding 100 non-dominated solutions, NSGA-II is the only heuristic with comparable graphical results to our technique, and presenting slightly better values for GD and S than those of MOPSO-*fs*, although there is not a high statistical difference between these results, according to the Student's *t-test*.

For the third test function, when finding 10 non-dominated solutions, again MOPSO presents better values for the S metric and even the GD metric (see Table 3.3). This is because all its solutions are found in a very small portion of the Pareto front, as can be seen by the very small D value that it presents (see Table 3.3 and Figure 3.15). The small D values of MOPSO have a very high significant statistical difference on the Student's *t-test*, over the average high D values of MOPSO-*fs*. In metrics where NSGA-II seem to perform slightly better than MOPSO-*fs*, it does not exist a high statistical difference between the pair of results, according to the Student's *t-test*; on the other hand, there is a significant statistical difference between results, with at least a 95% confidence level, when MOPSO-*fs* outperforms NSGA-II. As for general performance against PAES for this problem, MOPSO-*fs* performs much better.

For the fourth test function, when finding 10 non-dominated solutions, NSGA-II and PAES are the ones that show similar graphical performance to MOPSO-*fs* (Figure 3.16, 10 non-dominated solutions), but as can be seen in the statistical results our technique outperforms the way it spaces its solutions found (see S values in Table 3.4, for 10 non-dominated solutions). MOPSO again has problems to spread its solutions evenly when using a small population number. When looking for 100 non-dominated solutions, all of the techniques perform relatively well statistically speaking; although graphically, MOPSO-*fs* solutions look slightly more evenly distributed than solutions found by the other techniques, see Figure 3.16.

For test functions 1, 2, 3 and 4, and when finding 100 non-dominated solutions, statistics show that all methods are very competitive for all the test functions. With regards on graphical comparisons MOPSO-*fs* seems to perform slightly better in terms of distribution and extension over the Pareto front, again due to its fitness sharing mechanism. In Figure 3.13 we can see that all the techniques (except PAES) reach the Pareto Front but the one that has better distribution of solutions is our technique (with a lower S value in Table 3.1). The third test function presents three disjointed Pareto fronts, for the two most inferior fronts our technique presented a very good distribution of solutions, while having a bit more difficulty on the top most front, but as can be seen the rest of the heuristics present difficulties in at least one of the fronts, and in general they don't present evenly distributed solutions. In Figures 3.14 and 3.16, we can notice that the MOPSO-*fs* solutions are more evenly distributed than the solutions given for the rest of the techniques.

The second part of our experiments involved test functions with three objectives (test function 5, 6 and 7), which are more difficult to solve. All heuristics used here, when using small populations, tend to find (or sometimes not even find) bad approximation sets to the Pareto front. Due to this reason, we will discuss only results involving 100 non-dominated solutions.

Test function 5, on its graphical comparison between heuristics, Figure 3.17, shows the difficulty that all of them (except NSGA-II) have to converge to the hyper-plane. NSGA-II, is for this test by far, the heuristic that shows best results graphically and statistically 3.5.

For test function 6, plotting results by heuristics MOPSO-*fs* and NSGA-II show better performance, as we can see on Figure 3.18; is specially distinctive how MOPSO-*fs* has distributed its solutions uniformly along the Pareto front surface. This can be confirmed by looking at the statistics shown on Table 3.6, which support that MOPSO-*fs* performs better than the other heuristics, for this test function.

In our last test function, test function 7, we can see that graphically MOPSO-*fs* has problems to converge to the four hyper-planes that this test function has (see Figure 3.19). On one hand, MOPSO-*fs* manages to maintain distributed sub-populations along the different disconnected Pareto-optimal regions, which even look slightly more distributed than the other heuristics; but on the other hand, NSGA-II and PAES manage to better converge to the hyper-planes. Because of this reason, we presume is why statistically, NSGA-II and PAES outperform MOPSO-*fs* (see Table 3.7).

3.5 Summary

From the experiments shown, we can say that we have built a competitive heuristic using a particle swarm optimizer and fitness sharing to help it to deal with multi-objective problems. Our experiments have shown us that the PSO with fitness sharing is specifically superior when finding a small number of non-dominated solutions, in functions with two objective functions. We attribute this to the mechanism that selects particles to enter into the repository (where we store the non-dominated solutions). The implementation of this mechanism leads to: first, filling the repository with non-dominated solutions; and second, to discard solutions from getting into the repository with lower fitness sharing. In that way, we are ensuring convergence, by giving priority to get into the repository non-dominated particles; and ensuring diversity, by removing particles in highly populated areas.

Still some work needs to be done in order to improve this algorithm; an automatic way to find an appropriate value of σ_{share}–which until now has been empirically tuned–needs to be found. Work in this area is presented in the following Chapter.

Chapter 4

Particle Swarm Optimisation and Auto-Fitness Sharing to Solve Multi-Objective Problems

In the previous Chapter we introduced a modification to the PSO technique to make it capable to deal with multi-objective optimisation problems. The heuristic made use of fitness sharing to distribute its solutions along the Pareto front. Empirical studies showed that the technique is very effective. However, one of its weaknesses is the correct setting of a parameter used by fitness sharing, which has to be tuned manually and according to the problem that it is trying to solve. This is a general drawback when using fitness sharing.

In this Chapter we present a solution to this drawback, we will show the benefits of incorporating auto-adaptation for this parameter in our technique, and how this improves its performance.

4.1 Proposed Approach

The modification presented in the previous Chapter–to the original PSO heuristic, to make it capable to handle MOPs–made use of fitness sharing. The use of fitness sharing was done over the objective space; having in mind that we wanted to spread solutions along the Pareto front, and that we wanted to help to our algorithm to maintain diversity between solutions (see Figure 3.1). A schematic showing the flow of the algorithm can be seen in Figure 3.2.

As we have mentioned, in the proposed approach from the previous Chapter, the setting of the σ_{share} parameter used by fitness sharing was manually adjusted for every different problem we wanted to solve, and according to the number of solutions we were aiming to find over a run. To avoid this, we now propose a way to automatically adjust the value of the σ_{share} parameter; in words, the value of this parameter represents the radius distance that we would like solutions to remain apart from each other.

The algorithm proposed here, in structure, is similar to the one presented in the previous Chapter (see Section 3.1); the main difference between the two relies on how σ_{share} is calculated for the use of fitness sharing.

4.1.1 Auto-Fitness Sharing

Trying to overcome one disadvantage of our previously proposed technique, we are propounding a way to calculate a value for σ_{share} automatically. Our proposed self-adaptive fitness sharing method, updates the σ_{share} value at the end of every generation. We do this by looking for the farthest solutions found so far in the objective space and then calculating the Euclidean distance between them. With this distance we calculate σ_{share} as:

$$\sigma_{share} = \frac{D}{n} \tag{4.1}$$

where D is the distance between the two farthest solutions in the repository, and n is the number of solutions non-dominated found so far, in other words, n is the number of solutions in the repository.

In this way we will be encouraging to maintain the solutions found in our repository as apart, or diverse, as possible but trying to equally, or symmetrically, disperse them over the resources, in this case the Pareto front.

Is worth mentioning that previously Fonseca and Fleming [42] proposed an alternative way to set the σ_{share} parameter. Their method (which was applied to genetic algorithms) tries to cover the maximum hyper-volume that could occur between the more distant solutions, whereas we try to cover the line between the farthest solutions.

4.2 Test and Comparison

To test and compare this new approach we will use the same metrics and test functions, introduced and employed, in the previous Chapter (see Section 3.2).

The experiments made to test the new proposal were similar to the ones made in the previous Chapter, we compared our approach against: MOPSO [20], NSGA-II [25] and PAES [62][1].

Similar to our previous experiments, in the first part, when using test functions with two-objectives (test function 1, 2, 3 and 4), we have performed two sets of experiments. In the first set of experiments we have set the heuristics to find only 10 non-dominated solutions per run, and in the second set 100 non-dominated solutions. The second part of our experiments, test functions with three-objectives (test function 5, 6 and 7), we performed just one set of experiments involving 100 non-dominated solutions.

To allow a fair comparison between all the heuristics, they performed the same amount of evaluations of the objective function in each test. We performed two set of experiments, in the first one our objective was to find at least 10 solutions, and in the second one at least 100 solutions. To compare and obtain statistics for each test function, we performed 30 runs for each technique used.

The parameters used by MOPSO, NSGA-II and PAES for all the test functions were:

MOPSO used a mutation rate of 0.05, 30 divisions for its adaptive grid,

[1] Source code for these heuristics was written by their respective authors and obtained from [16].

and a real number representation; for the first set of experiments a population of 10 particles and a repository size of 10 particles, for the second set of experiments a population of 100 particles and repository size of 100 particles.

NSGA-II used a crossover rate of 0.8, a mutation probability of $1/x$, where x is the number of variables for the given problem, and a real number representation with tournament selection; for the first set of experiments a population of 10 individuals, and for the second set of experiments a population of 100 individuals.

PAES used a depth value of 5, a mutation probability of $1/L$, where L refers to the length of the chromosomic binary string, that encodes the decision variables; for the first set of experiments an archive size of 10 individuals, and for the second an archive size of 100.

MOPSO-auto-fs used a population of 10 particles and a repository size of 10 particles for the first set of experiments, and for the second a population of 100 particles and a repository size of 100 particles was used.

Values for both MOPSOs, in the calculation of the velocity formula, were of 0.4 for the w, and 1.0 for c_1 and c_2 coefficients.

The value of the number of evaluations for each of the test functions will be specified in each test function section.

The runs we choose to plot, were selected based on the median given by one of the metrics or indicators used; e.g., out of the 30 runs for each heuristic, the one closer to the median of the GD metric, or the I_H^- indicator, was the one plotted for that particular test function.

4.2.1 Test function 1

The first test function is the one proposed by Fonseca [43] (equation 3.14, here referred as FON).

For this problem all the heuristics were set to evaluate $30,000$ times the objective function. In Table 4.1 we can observe the statistical results obtained when comparing the four different approaches. In Figure 4.1 we can observe a graphical comparison of the results for the four different techniques.

4.2.2 Test function 2

Our second test function, is a maximization problem proposed by Poloni [24, pg. 328] (named POL, see equation 3.15).

For the second test function all the heuristics performed $10,000$ evaluations to the objective function. Table 4.2 has the statistical values that we obtained from measuring the results, and Figure 4.2 has the graphical representations of the set of non-dominated solutions found by each of the heuristics.

4.2.3 Test function 3

Kursawe's [70] (KUR, see equation 3.16) is our third test function.

For this test function the number of evaluations was $30,000$ times. Table 4.3 contains the statics of the comparison for this test function between the 4 heuristics. Figure 4.3 has a graphical representation of the set of non-dominates solutions for all of the techniques.

10 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.0036792	0.0013595	0.012794	0.017031
	S.D.	0.001643	0.00034466	0.0044081	0.006975
	Med.	0.0031247	0.0013507	0.011513	0.01769
	t-test	—	3.2516E-10 ∗	3.2726E-15 ∗	1.454E-14 ∗
S	*Avg.*	0.033754	0.010122	0.092068	0.063584
	S.D.	0.0091986	0.00726	0.023389	0.078061
	Med.	0.035993	0.0093974	0.091994	0.029477
	t-test	—	6.848E-16 ∗	2.0909E-18 ∗	0.042086 ∗
D	*Avg.*	1.2964	0.15709	1.3883	0.69795
	S.D.	0.058421	0.076672	1.5992E-05	0.080328
	Med.	1.3117	0.17716	1.3883	0.68558
	t-test	—	8.7133E-56 ∗	5.7275E-12 ∗	2.7552E-39 ∗
100 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.00076519	0.0007684	0.00079085	0.0041403
	S.D.	2.8796E-05	2.7626E-05	2.5159E-05	0.0015396
	Med.	0.0007614	0.0007675	0.00078717	0.0045956
	t-test	—	0.66119	0.00052068 ∗	2.3172E-17 ∗
S	*Avg.*	0.0042346	0.0086783	0.0076931	0.0071636
	S.D.	0.00058214	0.00067753	0.00064243	0.013006
	Med.	0.0042258	0.0086366	0.0077907	0.0029712
	t-test	—	9.5032E-35 ∗	1.1213E-29 ∗	0.22283
D	*Avg.*	1.3801	1.3645	1.3883	0.75282
	S.D.	0.0050889	0.009549	0.00017547	0.095179
	Med.	1.3794	1.3644	1.3883	0.71525
	t-test	—	1.0589E-10 ∗	2.4454E-12 ∗	2.0903E-41 ∗

Table 4.1: This table shows statistical results (average, standard deviation, median) of the metrics used over test function 1 (FON). It also shows results of a Student's t-test study, when comparing MOPSO-auto-*fs* against the other three heuristics. [∗] Means a significant difference between results, with at least a 95% confidence level.

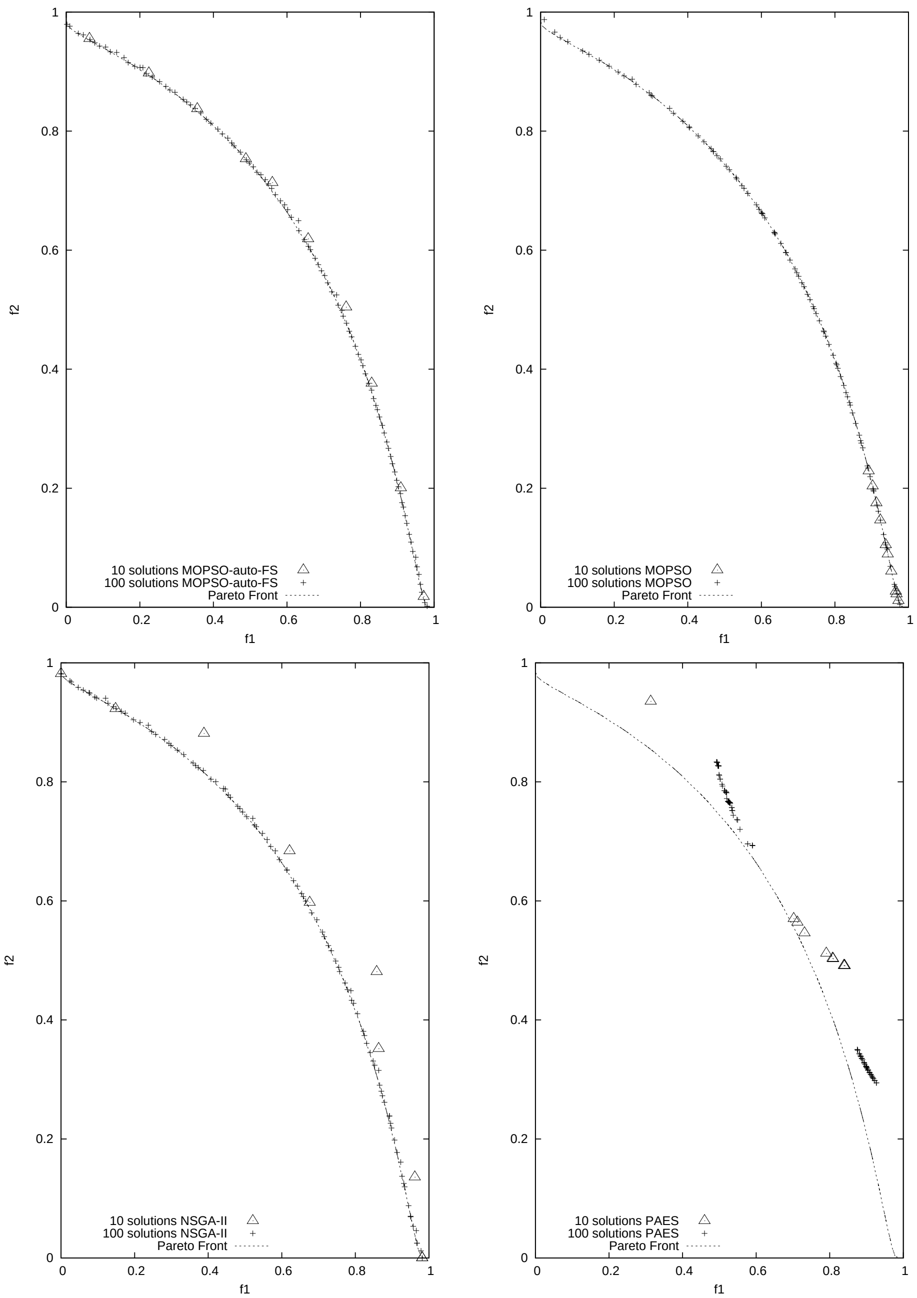


Figure 4.1: This graphical results were obtained from test function 1 (FON).

		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
10 Non-Dominated Solutions					
GD	*Avg.*	0.087159	0.08691	0.14632	0.1122
	S.D.	0.14092	0.18528	0.16751	0.21023
	Med.	0.0083992	0.0076148	0.034161	0.01511
	t-test	—	0.99535	0.1442	0.58996
S	*Avg.*	1.079	1.2302	1.3676	2.7549
	S.D.	0.36544	1.9859	0.5077	2.6771
	Med.	1.1713	0.35563	1.3752	1.8797
	t-test	—	0.68328	0.014249 ∗	0.0012353 ∗
D	*Avg.*	27.455	12.405	30.572	21.886
	S.D.	6.2767	10.281	1.356	8.1413
	Med.	29.386	5.7877	29.562	25.306
	t-test	—	5.3681E-09 ∗	0.010111 ∗	0.0043642 ∗
100 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.0063795	0.010097	0.0049512	0.0061743
	S.D.	0.010695	0.016192	0.011131	0.010138
	Med.	0.0021954	0.0023413	0.0020281	0.0025725
	t-test	—	0.29838	0.61424	0.93947
S	*Avg.*	0.13941	0.13073	0.094002	0.3265
	S.D.	0.075268	0.064069	0.0074151	0.54217
	Med.	0.11202	0.1116	0.09463	0.16132
	t-test	—	0.63213	0.0017137 ∗	0.066246
D	*Avg.*	29.912	30.003	29.745	20.505
	S.D.	0.96717	1.1806	0.70157	8.2331
	Med.	29.551	29.507	29.564	14.933
	t-test	—	0.74381	0.44861	5.9995E-08 ∗

Table 4.2: This table shows statistical results (average, standard deviation, median) of the metrics used over test function 2 (POL). It also shows results of a Student's t-test study, when comparing MOPSO-auto-*fs* against the other three heuristics. [∗] Means a significant difference between results, with at least a 95% confidence level.

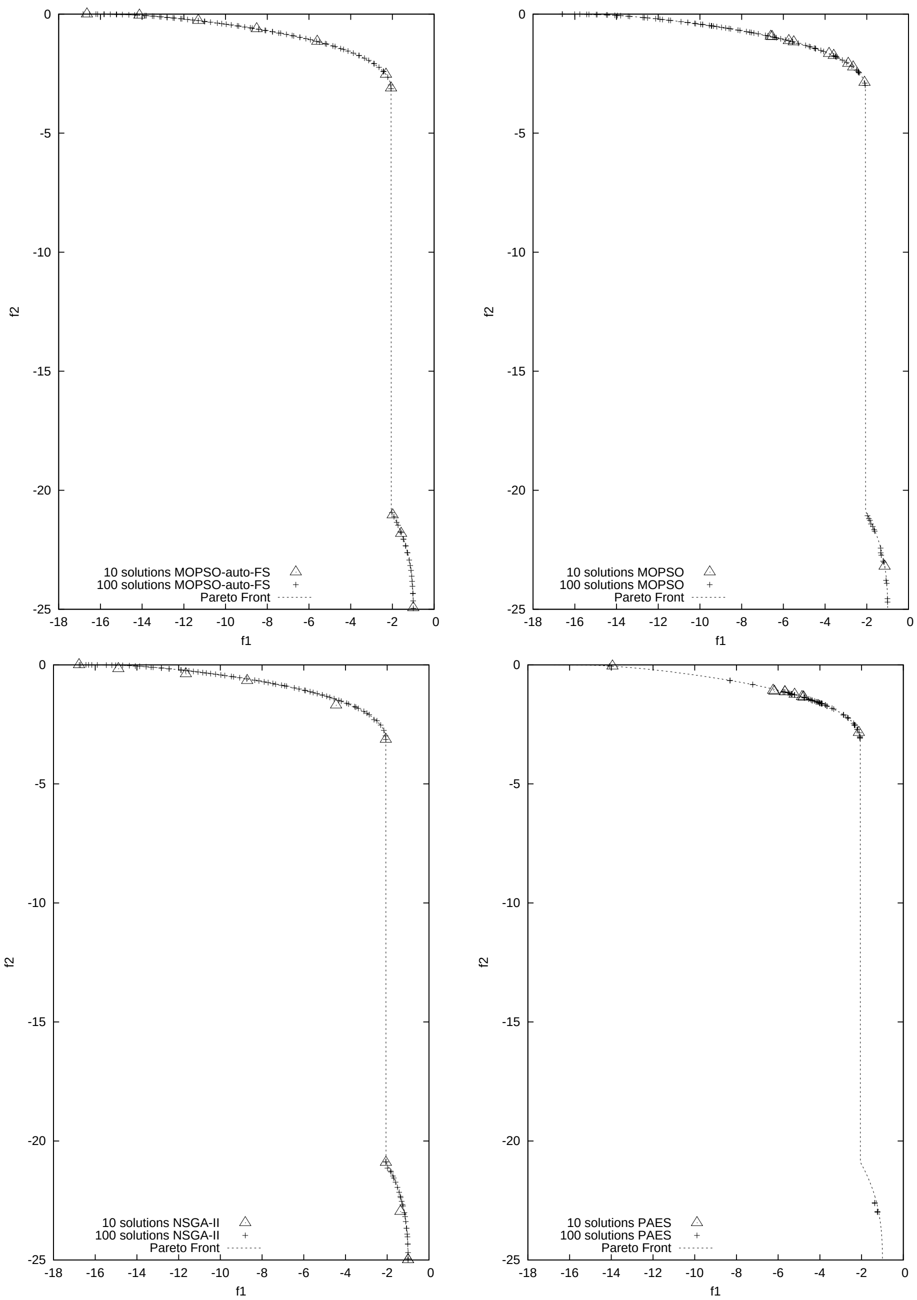


Figure 4.2: This plots correspond to the results obtained from function 2 (POL).

10 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.034069	0.0053432	0.070042	0.064137
	S.D.	0.073579	0.001036	0.050364	0.14594
	Med.	0.011116	0.0052896	0.075988	0.012062
	t-test	—	0.036731 ∗	0.031081 ∗	0.31781
S	*Avg.*	0.27056	0.059482	0.41547	1.0166
	S.D.	0.081931	0.053412	0.38433	0.47327
	Med.	0.25257	0.041538	0.26856	0.88527
	t-test	—	4.4019E-17 ∗	0.048035 ∗	8.6734E-12 ∗
D	*Avg.*	12.003	0.87372	3.2826	9.8123
	S.D.	1.3487	0.6596	1.44	2.1924
	Med.	12.513	0.68929	3.0714	10.037
	t-test	—	2.7232E-44 ∗	5.2401E-32 ∗	1.8911E-05 ∗
100 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.0051145	0.0042765	0.0046214	0.011573
	S.D.	0.0011625	0.00039943	0.0034957	0.016708
	Med.	0.0048178	0.0041994	0.0036337	0.0036493
	t-test	—	0.00043157 ∗	0.4664	0.038996 ∗
S	*Avg.*	0.077357	0.088236	0.067162	0.19149
	S.D.	0.016156	0.020342	0.035135	0.057394
	Med.	0.080664	0.096998	0.055365	0.18862
	t-test	—	0.025438 ∗	0.15416	5.2259E-15 ∗
D	*Avg.*	12.947	12.91	11.03	12.14
	S.D.	0.030872	0.033611	0.50698	1.4434
	Med.	12.945	12.913	11.025	12.717
	t-test	—	4.6771E-05 ∗	1.9535E-28 ∗	0.003325 ∗

Table 4.3: This table shows statistical results (average, standard deviation, median) of the metrics used over test function 3 (KUR). It also shows results of a Student's t-test study, when comparing MOPSO-auto-*fs* against the other three heuristics. [∗] Means a significant difference between results, with at least a 95% confidence level.

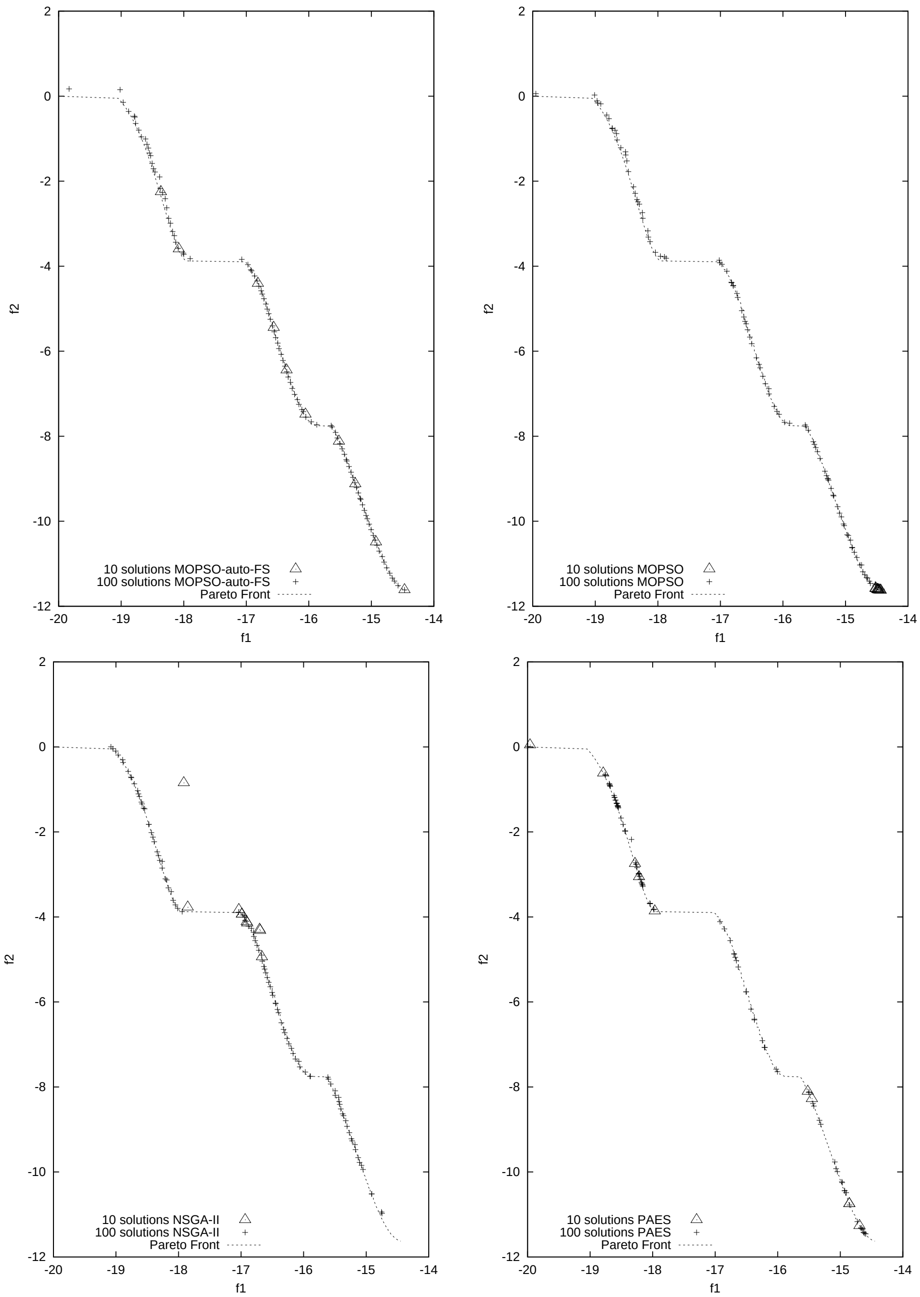


Figure 4.3: This plots correspond to the results obtained from test function 3 (KUR).

4.2.4 Test function 4

Our fourth test function, proposed by Deb [23] (DEB, see equation 3.17).

For the fourth test function each of the heuristics performed $5,000$ evaluations to the objective function. Table 4.4 and Figure 4.4 correspond to statistics and graphical representations to this problem, respectively.

4.2.5 Test function 5

DTLZ1 by Deb et al. [28, 29] (see equation 3.18) is our fifth test function. For this test function we performed $4,000$ evaluations. Table 4.5 contains the statics of the comparison for this test function between the 4 heuristics. Figure 4.5 has a graphical representation of the set of non-dominates solutions for all of the techniques. For this problem, the indicators I_H^- and I_{R3}^1 used the vector $(14, 14, 14)$ as a *nadir* point; the indicator I_{R3}^1, for ρ used a value of 0.01, for s a value of 30, and as an ideal vector $(0, 0, 0)$. The reference set used in here can be found in [16].

4.2.6 Test function 6

Our next test function proposed by Deb et al. [28, 29], is called DTLZ2 (equation 3.19). For this problem all the heuristics were set to evaluate 500 times the objective function. In Table 4.6 we can observe the statistical results obtained when comparing the four different approaches. In Figure 4.6 we can observe a graphical comparison of the results for the four different techniques. For this function the *nadir* reference point for indicators I_H^- and I_{R3}^1 $(1.5, 1.5, 1.5)$ was used; indicator I_{R3}^1 used a ρ value of 0.01, an s of 30,

10 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.029259	0.0032553	0.001822	0.10233
	S.D.	0.098923	0.0073799	0.0029862	0.2575
	Med.	0.0016218	0.00066509	0.00077624	0.00095592
	t-test	—	0.15643	0.13434	0.15217
S	*Avg.*	0.04491	0.0169	0.11093	0.32537
	S.D.	0.013136	0.019832	0.051593	0.38868
	Med.	0.044685	0.008917	0.11443	0.20455
	t-test	—	2.4483E-08 ∗	6.5541E-09 ∗	0.00021431 ∗
D	*Avg.*	1.5666	0.26749	1.4802	2.4641
	S.D.	0.098179	0.27643	0.54473	2.036
	Med.	1.5817	0.16241	1.6903	1.6786
	t-test	—	4.7247E-32 ∗	0.39626	0.019063 ∗
100 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
GD	*Avg.*	0.0019587	0.00029629	0.00033131	0.01264
	S.D.	0.0035372	3.799E-05	2.2599E-05	0.024335
	Med.	0.00043837	0.00029772	0.00033063	0.00043685
	t-test	—	0.012632 ∗	0.014518 ∗	0.020666 ∗
S	*Avg.*	0.018926	0.0096086	0.0073065	0.10663
	S.D.	0.031127	0.0022829	0.00066919	0.21121
	Med.	0.0088469	0.0094451	0.0074024	0.017581
	t-test	—	0.10744	0.045489 ∗	0.028259 ∗
D	*Avg.*	1.8367	1.6885	1.6902	2.7142
	S.D.	0.32989	0.0022992	0.00010657	2.1148
	Med.	1.6962	1.6893	1.6902	1.7157
	t-test	—	0.016875 ∗	0.01815 ∗	0.02855 ∗

Table 4.4: This table shows statistical results (average, standard deviation, median) of the metrics used over test function 4 (DEB). It also shows results of a Student's *t*-test study, when comparing MOPSO-auto-*fs* against the other three heuristics. [∗] Means a significant difference between results, with at least a 95% confidence level.

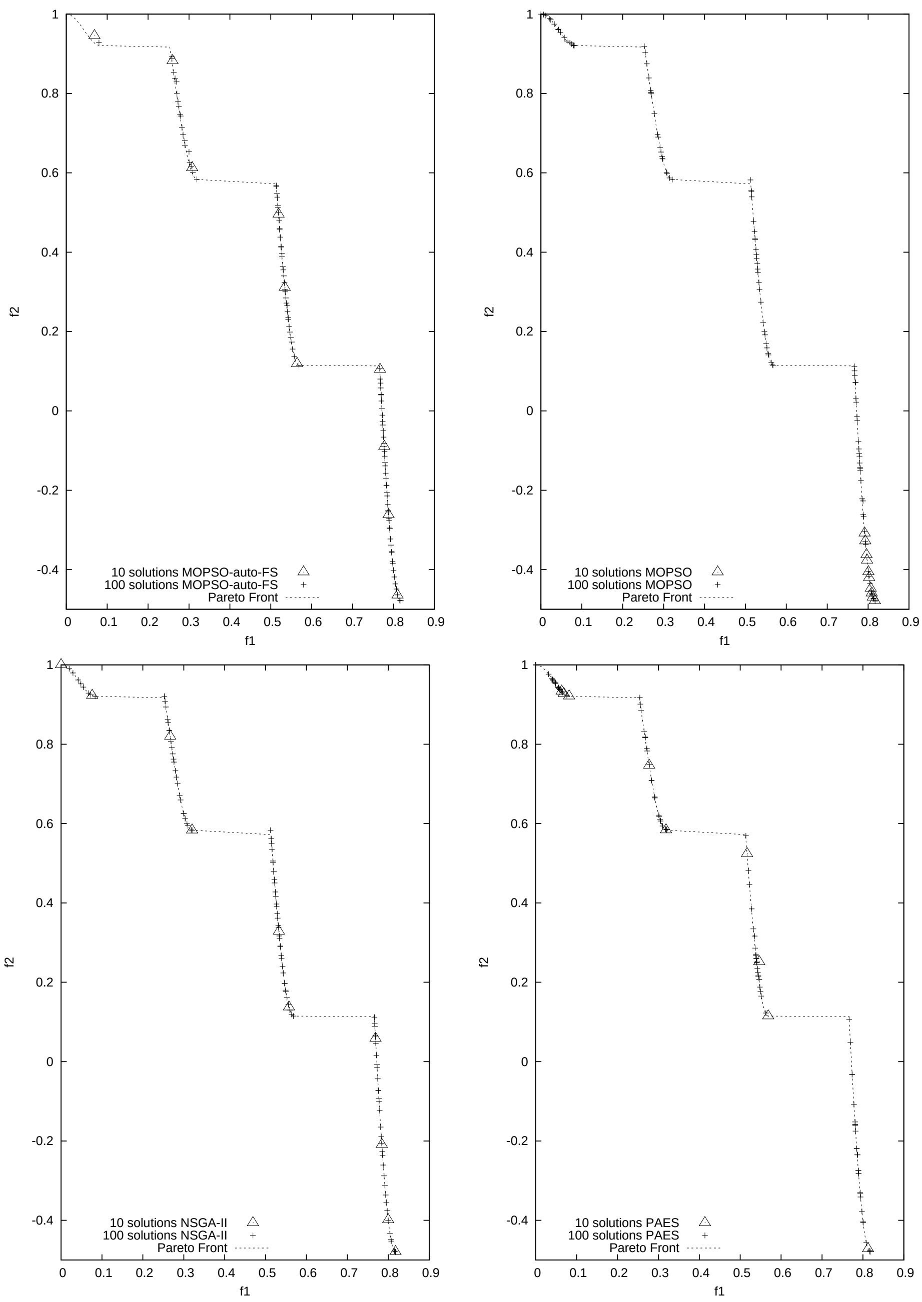


Figure 4.4: This plots correspond to the results obtained from test function 4 (DEB).

100 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
I_H^-	*Avg.*	24.64657	31.07963	-1.98463	11.99605
	S.D.	43.69592	79.80929	0.00077	15.69039
	Med.	9.31275	2.53136	-1.98474	5.62067
	t-test	—	0.69999	0.00148 $*$	0.14101
$I_{\epsilon+}^1$	*Avg.*	1.09689	0.70090	0.05295	1.16275
	S.D.	0.62378	0.39199	0.00602	0.60724
	Med.	1.08798	0.63675	0.05210	1.02028
	t-test	—	0.00465 $*$	0.00000 $*$	0.68017
I_{R3}^1	*Avg.*	0.01322	0.00945	0.00012	0.01200
	S.D.	0.00850	0.00852	0.00001	0.00640
	Med.	0.01253	0.00724	0.00011	0.01054
	t-test	—	0.09143	0.00000 $*$	0.53306

Table 4.5: This table shows statistical results (average, standard deviation, median) of the metrics used over test function DTLZ1. It also shows results of a Student's t-test study, when comparing MOPSO-auto-*fs* against the other three heuristics. [$*$] Means a significant difference between results, with at least a 95% confidence level.

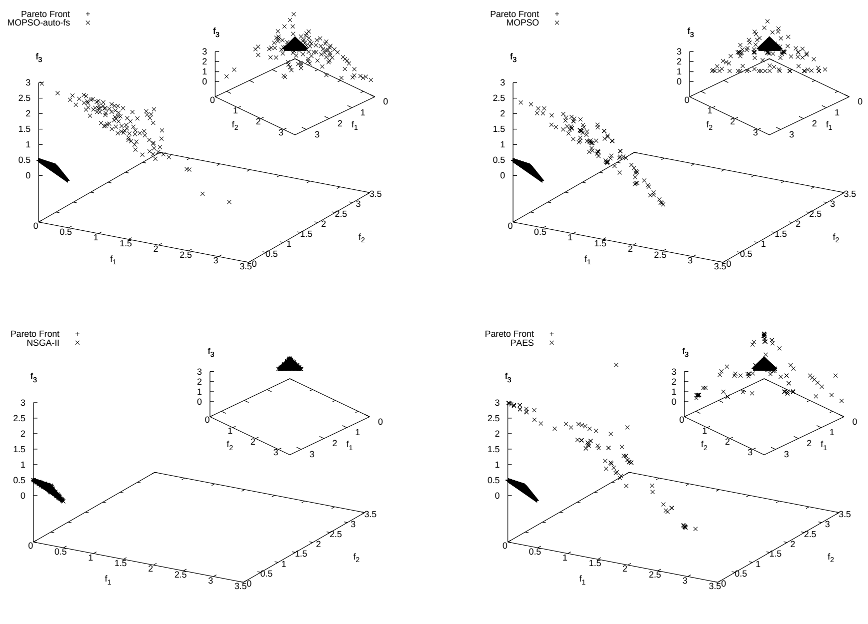


Figure 4.5: This plots correspond to the results obtained from test function 5 (DTLZ1).

100 Non-Dominated Solutions					
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
I_H^-	*Avg.*	0.04330	1.27678	0.06095	0.10431
	S.D.	0.00928	0.08622	0.00898	0.03535
	Med.	0.04077	1.25672	0.06015	0.09761
	t-test	—	0.00000 $*$	0.00000 $*$	0.00000 $*$
$I_{\epsilon+}^1$	*Avg.*	0.11216	0.78940	0.12693	0.16343
	S.D.	0.01849	0.04798	0.02283	0.03010
	Med.	0.11125	0.78230	0.12074	0.15527
	t-test	—	0.00000 $*$	0.00783 $*$	0.00000 $*$
I_{R3}^1	*Avg.*	-0.00025	0.12635	-0.00002	0.00282
	S.D.	0.00080	0.01024	0.00048	0.00237
	Med.	-0.00050	0.12451	-0.00020	0.00203
	t-test	—	0.00000 $*$	0.18354	0.00000 $*$

Table 4.6: This table shows statistical results (average, standard deviation, median) of the metrics used over test function DTLZ2. It also shows results of a Student's t-test study, when comparing MOPSO-auto-*fs* against the other three heuristics. [$*$] Means a significant difference between results, with at least a 95% confidence level.

and the ideal vector $(0, 0, 0)$.

4.2.7 Test function 7

Deb's et al. DTLZ7 [28, 29] (see equation 3.20) is our last test function. For this test function the number of evaluations was $1,000$ times. Table 4.7 contains the statics of the comparison for this test function between the 4 heuristics. Figure 4.7 has a graphical representation of the set of non-dominated solutions for all of the techniques. The values for the *nadir* vector for indicators I_H^- and I_{R3}^1 are $(1, 1, 11.5)$, the I_{R3}^1 indicator used a ρ of 0.01, a s value of 30, and an ideal vector of $(0, 0, 0)$.

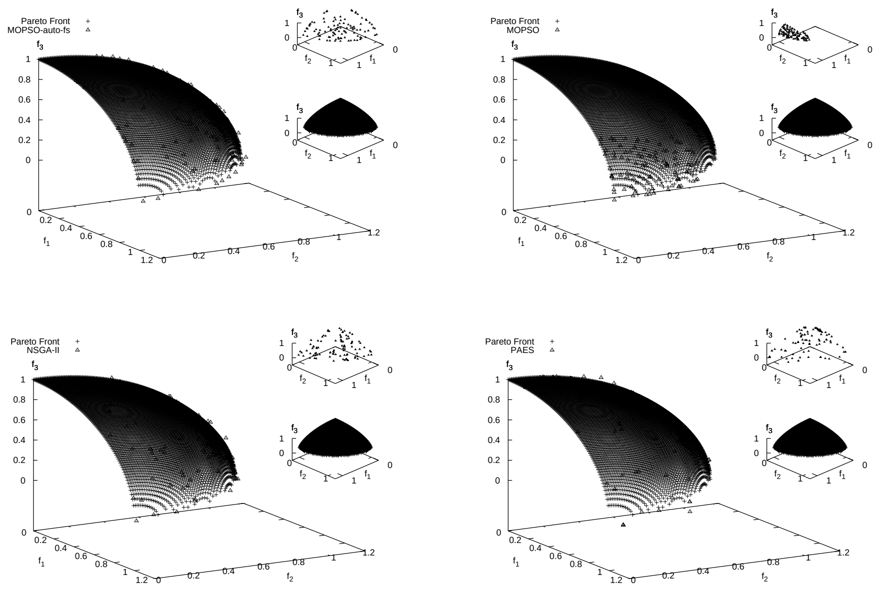


Figure 4.6: This graphical results were obtained from test function 6 (DTLZ2).

		100 Non-Dominated Solutions			
		MOPSO-*fs*	**MOPSO**	**NSGA-II**	**PAES**
I_H^-	*Avg.*	2.19427	5.03304	0.19226	0.45590
	S.D.	0.48988	1.50527	0.13602	0.33693
	Med.	2.11320	5.04651	0.13037	0.30719
	t-test	—	0.00000 $*$	0.00000 $*$	0.00000 $*$
$I_{\epsilon+}^1$	*Avg.*	3.15872	3.49238	0.41708	0.38796
	S.D.	0.88259	0.73488	0.48013	0.21332
	Med.	3.13239	3.53521	0.18262	0.32898
	t-test	—	0.11699	0.00000 $*$	0.00000 $*$
I_{R3}^1	*Avg.*	0.08201	0.21779	0.00645	0.00942
	S.D.	0.02306	0.08404	0.00915	0.00717
	Med.	0.08100	0.19066	0.00203	0.00648
	t-test	—	0.00000 $*$	0.00000 $*$	0.00000 $*$

Table 4.7: This table shows statistical results (average, standard deviation, median) of the metrics used over test function 7 (DTLZ7). It also shows results of a Student's t-test study, when comparing MOPSO-auto-*fs* against the other three heuristics. [$*$] Means a significant difference between results, with at least a 95% confidence level.

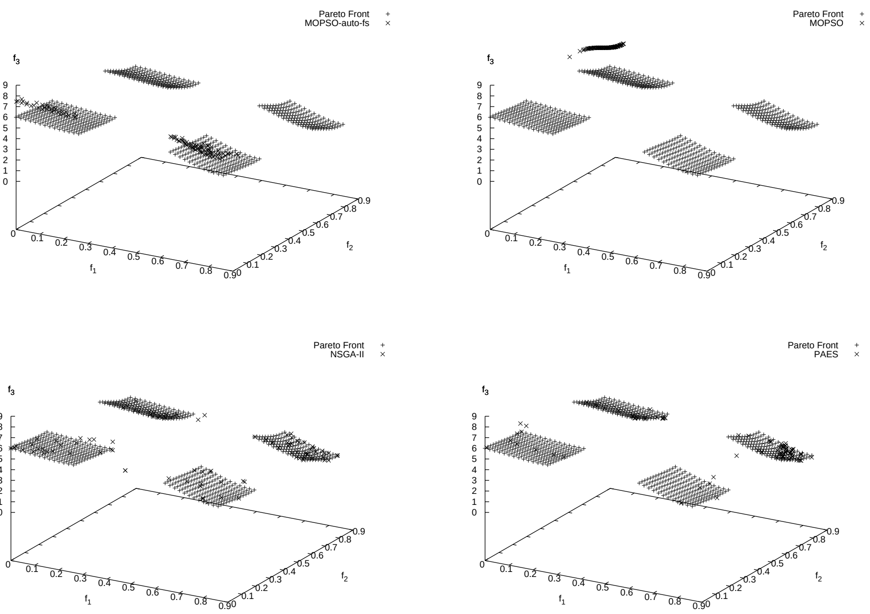


Figure 4.7: This plots correspond to the results obtained for test function 7 (DTLZ7).

4.3 Discussion of Results

In the graphical representations of the results obtained by all the heuristics for all the test functions with two-objectives (test functions 1, 2, 3 and 4, in Figures 4.1, 4.2, 4.3 and 4.4, respectively), we can observe that MOPSO-auto-*fs* obtains better results than the other techniques. We attribute this to our implementation of fitness sharing and the way its σ_{share} value is automatically updated along the life cycle of the algorithm.

From a statistical point of view, for the first test function, and relating experiments with 10 non-dominated solutions, MOPSO has the better values for the GD and S metric (as can be seen in Table 4.1), but this is due to the fact that MOPSO is concentrating all its solutions in a very small portion of the Pareto front (see small D value for MOPSO, and MOPSO with 10 non-dominated solutions in Figure 4.1), not necessarily due to a better set of dispersed solutions. NSGA-II is the only heuristic that graphically (Figure 4.1) shows comparable results to MOPSO-auto-*fs*, but statistically our technique shows better results in approaching the Pareto front and also in dispersing the solutions (smaller GD and S values in Table 4.1), for both set of experiments with 10 and 100 non-dominated solutions. We claim that this statistical results are significantly different with a 95% of confidence level, which can be supported by the Student's *t-test* values reported in Table 4.1.

For the second test function, a very similar situation as the presented with the first test function can be observed. MOPSO presents better values for the GD and S metric (see Table 4.2, and Figure 4.2), but this is due to a concentration of all its solutions found in a very small portion of the

Pareto front. NSGA-II again is the only heuristic with comparable results to our technique, here MOPSO-auto-*fs* presents better values for GD and S when looking for 10 non-dominated solutions (Table 4.2), And is just slightly behind NSGA-II which out when looking for 100 non-dominated solutions.

For the third test function, again MOPSO presents better values for the GD and S metric, in the set of 10 non-dominated solution experiments. This is due to a concentration of all its solutions found in a very small portion of the Pareto front, as can be seen by the very small D value that it presents (see Table 4.3, and Figure 4.3). When searching for no more than 10 non-dominated solutions, MOPSO-auto-*fs* performs better against NSGA-II and PAES. When looking for 100 non-dominated solutions, MOPSO-auto-*fs*, MOPSO and NSGA-II perform relatively the same, apart from PAES which gets slightly worst GD and S metrics results.

For the fourth test function, when looking for only 10 non-dominated solutions, MOPSO again is not capable of spreading its solutions, and NSGA-II along with PAES are the ones that show similar performance to the one of MOPSO-auto-*fs* (see Figure 4.4); but as can be seen in the statistical results, our technique performs slightly better on the S metric, by spreading better its solutions along the Pareto front (see Table 4.4). For 100 non-dominated solutions, all of the heuristics perform relatively well, although NSGA-II performs slightly better than the rest, as can be seen in Table 4.4.

Test function 5 (DTLZ1) is a problem with three objectives to be minimised, as are the two following test function problems. In the graphical representation for this test function, we can see how hard it is for most of the heuristics to converge to the Pareto front (see Figure 4.5); in fact, NSGA-II is the only one capable to reach the hyper-plane. As a result, all of the heuristics, apart from NSGA-II, perform quite badly in the quality indicators, see Table 4.5.

In test function 6 (DTLZ2), graphically in Figure 4.6, we can see that although most of the heuristics perform better, than in the previous test function 5, MOPSO is not quite capable to distribute its solutions along the entire surface of the Pareto front. On the other hand, MOPSO-auto-*fs*, is the technique which outperforms the rest; as can be seen, its solutions spread quite uniformly along the surface, and the performance indicators show us that MOPSO-auto-*fs*, compared to the other three, performs better than them (see Table 4.6).

In our last test function, 7 (DTLZ7), we can see that both MOPSOs have problems to converge to the four different hyper-planes (Figure 4.7). Although MOPSO-auto-*fs* manages to distribute its solutions into sub-populations, in different Pareto-optimal regions, finds it hard to converge to the Pareto front. NSGA-II and PAES are the ones which perform the best for this test problem. Although the sets of results are incomparable between them, as two indicators are contradicting on the performance between them; indicators I_H^- and I_{R3}^1 give a better performance to NSGA-II, and indicator $I_{\epsilon+}^1$ gives a better performance to PAES (see Table 4.7).

4.4 Summary

For what we can gather from the experiments, the technique here introduced, MOPSO-auto-*fs* successfully performed against the other state-of-the-art techniques, specially when looking for a small amount of solutions and on problems with only two-objective functions (see Section 4.3). We believe we have created a competitive technique to tackle multi-objective problems, which performs in most of the cases better than the rest, by better distributing the solutions it finds, along the Pareto front; and specially, as we mentioned, when looking for a small amount of solutions and on problems which have two objectives. Also we are aware that, the technique has some problems to converge to the Pareto front with more difficult test problems (as was the case with test function 5 and 7), and we will further discuss this in our conclusions.

Chapter 5

Particle Swarm Optimisation on Dynamic Multi-Objective Optimisation Problems

In this Chapter we want to show the potential that the PSO heuristic has to handle multi-objective dynamic environments.

The heuristic will be tested and analysed with dynamic multi-objective problems taken from the specialised literature.

5.1 Dynamic Multi-Objective Optimisation and PSO

In multi-objective optimisation, whether is a static or a dynamic environment, we always deal with two spaces: variable space and objective space (see Section 2.1.2). This complexity already established, escalates even more

	Variable Space	**Objective Space**
Type I	Change	No Change
Type II	Change	Change
Type III	No Change	Change
Type IV	No Change	No Change

Table 5.1: Types of problems in a dynamic multi-objective environment.

in dynamic problems because as Farina et al. [36] suggest, there are four possible ways in which a problem can dynamically change:

Type I: The Pareto optimal set (optimal decision variables) changes, whereas the Pareto optimal front (optimal objective values) do not change.

Type II: The Pareto optimal set (optimal decision variables) changes, as well as the Pareto optimal front (optimal objective values).

Type III: The Pareto optimal set (optimal decision variables) do not change, whereas the Pareto optimal front (optimal objective values) changes.

Type IV: Both the Pareto optimal set (optimal decision variables) and the Pareto optimal front (optimal objective values) do not change, although the problem can change.

These four types of problems are summarised in Table 5.1.

In Section 2.5, we have mentioned that PSO has had a successful incursion into the dynamic single-objective optimisation research area. Up to this date and to the best of our knowledge, we are unaware of any work done regarding PSO and dynamic multi-objective optimisation. Because of that, we are interested to know what are the capabilities of the heuristic in this area.

5.2 Proposed Approach

Our approach is based in the algorithm presented in Section 3.1, and later updated in Section 4.1.1, in fact it remains the same except for the way it deals when a change in the environment occurs.

The PSO heuristic has the implicit advantage that the swarm carries on over generations certain amount information. Information that at some point in time and under given circumstances might, or not, be always useful; by being useful, this information is what moves and helps to the particles to travel along the search space and eventually to find optima.

In a dynamic environment, when a change in the environment occurs, the information that particles carry with them may become invaluable, or even misleading in their search process. Particles might be trying to use information to fly towards an optimum that may no longer exist, in a new reshaped environment [34, chap. 19.1].

This is why when a change in the environment occurs, we have considered two simple reaction methods to adapt to the reshape of the environment:

(a) The memory of every particle is updated with the one that has the best location[1] between: (1) the current location of the particle, or (2) the one stored in its memory.

(b) The memory of every particle is reinitialised with the current position of the particle.

By using this two simple methods we believe that the PSO heuristic will be able to track any change in the environment. When using the first

[1]Here, best location is considered in terms of Pareto dominance.

method, we are giving the particles the ability to update information about the new environment, by using their current positions as new useful data, or by retaining the information stored in their memories; the latter case applies only when the information they have is still valuable under the new re-established conditions. The second method eliminates any stale information that could drive to the particles towards an old optimum, and forces them to forget it, by reseting their memories to where they were left.

The algorithm in its current form is not capable of detecting any change on the environment[2]. For that reason, we explicitly tell to our algorithm when a change in the environment has occurred; and at that particular time, to employ one of the two mechanisms proposed above.

5.3 Measuring and Testing Performance

Because we are interested to see if PSO is able to track optima we will be using the generational distance metric, introduced in Section 3.2. Every generation we will be measuring the distance that the external repository has from the real Pareto front.

5.3.1 Test Functions

Farina et al. [37] introduced a set of dynamic multi-objective optimisation test problems which we will use to study the performance of our technique.

The test problems used here have two objectives to be minimised, and have the following generic form:

[2]This extension is outlined as future work in Chapter 6.

$$\begin{aligned} &\text{Minimise} \quad f_1(\vec{x}) \\ &\text{Minimise} \quad f_2(\vec{x}) = g(\vec{x})h(f_1(\vec{x}), g(\vec{x})) \end{aligned} \tag{5.1}$$

Test function 1

This test problem is an illustration of a type I problem (see Section 5.1), the Pareto Front remains static but the variable space is the one which changes over time. It has two objectives to be minimised as in Equation 5.1:

$$\text{FDA1}: \begin{cases} f_1(\vec{x_I}) = x_1 \\ g(\vec{x_{II}}) = 1 + \sum_{x_i \in \vec{x_{II}}} (x_i - G(t))^2 \\ h(f_1, g) = 1 - \sqrt{\frac{f_1}{g}} \\ G(t) = \sin(0.5\pi t) \qquad t = \frac{1}{n_t} \left\lfloor \frac{\tau}{\tau_T} \right\rfloor \\ \vec{x_I} = (x_1) \in [0, 1] \qquad \vec{x_{II}} = (x_2, \ldots, x_n) \in [-1, 1] \end{cases} \tag{5.2}$$

Where $n = 20$, τ is the generation counter, τ_T is the number of generations for which t remains fixed, and n_t is the number of distinct steps in t. The values for τ_T and n_t will be specified later on.

Test function 2

This test problem is an illustration of a type III problem (see Section 5.1), the variable space is static, what changes over time is the Pareto Front, it changes from a convex to a non-convex shape, it has two objectives to be minimised and the problem has the generic form of Equation 5.1:

$$\text{FDA2m}: \begin{cases} f_1(\vec{x_I}) = x_1 \\ g(\vec{x_{II}}) = 1 + \sum_{x_i \in \vec{x_{II}}} x_i^2 \\ h(f_1, g) = 1 - (\frac{f_1}{g})^{H(t)} \\ H(t) = 0.75 + 0.7\sin(0.5\pi t) \qquad t = \frac{1}{n_t}\left\lfloor \frac{\tau}{\tau_T} \right\rfloor \\ \vec{x_I} = (x_1) \in [0,1] \qquad \vec{x_{II}} = (x_2, \ldots, x_n) \in [-1,1] \end{cases} \tag{5.3}$$

Where $n = 30$, τ is the generation counter, τ_T is the number of generations for which t remains fixed, and n_t is the number of distinct steps in t. The values for τ_T and n_t will be specified later on.

5.4 Test and Analysis

To test the algorithm we have made two set of experiments. In the first set we have fixed a small value for the τ_T variable, and a longer value for the second set. As τ_T regulates the longevity of a stable environment, we have performed two type of experiments in order to verify the stability of PSO over short and longer periods of time in dynamic environments.

For each set of experiments we used two types of methodologies to deal with a changing environment. These methods were described in Section 5.2.

The experiments that we performed are summarised in the following Table:

Experiment	Small τ_T		Big τ_T	
Method	(a)	(b)	(a)	(b)
Number of Executions	30	30	30	30

In summary, two sets of experiments were performed; on each of this, two methods were tried; and 30 runs were performed for each one.

The parameters for the PSO remained the same for all the experiments here performed, these parameters are: a population of 100 particles; an external repository for 100 particles; in the velocity formula (Equation 2.12), for the acceleration coefficients c_1 and c_2 a constant value of 0.494 was used, and for the inertia weight w a constant value of 0.4, were empirically set. The number of generations will be specified for each set of experiments.

5.4.1 Test function 1

For test function 1 (FDA1, Equation 5.2) we have used the following parameters: $n_t = 10$; as we have mentioned, two set of experiments were performed. The first set used a $\tau_T = 10$, and the second a $\tau_T = 50$. As we remember, τ_T is the number of generations for which the environment will remain stable.

Experiment Set 1 (Small τ_T)

The value of $\tau_T = 10$ for this experimental set, means that 10 is the number of generations for which the environment is going to remain stable. Because we want to show the PSO capabilities of tracking dynamic optima, we will follow its behaviour during one-hundred generations, which in turn will allow us to track its performance during 10 cycles of environmental change.

In Table 5.2 we show the average and standard deviation values, before and after a change in the environment occurs for ten environmental changes, given by our two methods (a) and (b). We have to remember that smaller

Environmental Change	**Average**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.184580	0.192641	0.702865	0.715224
2	0.058780	0.051877	0.217970	0.196400
3	0.039273	0.035685	0.132920	0.119871
4	0.043344	0.036656	0.130066	0.113194
5	0.043927	0.039169	0.130426	0.116721
6	0.039379	0.034548	0.116460	0.102835
7	0.029745	0.025353	0.094289	0.083042
8	0.023078	0.018622	0.065378	0.053301
9	0.017904	0.015150	0.043957	0.036104
10	0.013923	0.011661	0.028870	0.024505
Environmental Change	**Standard Deviation**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.064977	0.047546	0.128233	0.136796
2	0.020362	0.013879	0.087376	0.053920
3	0.009183	0.007730	0.033164	0.034192
4	0.008663	0.005539	0.026443	0.019418
5	0.010032	0.007547	0.024002	0.016684
6	0.008717	0.007792	0.020841	0.016192
7	0.006929	0.006819	0.022500	0.019190
8	0.005989	0.004477	0.016471	0.011923
9	0.004112	0.003505	0.008517	0.007257
10	0.003371	0.002490	0.006236	0.005390

Table 5.2: This table corresponds to statistical results obtained from test function 1 (FDA1). Data in this table was obtained by performing a change in the environment every 10 generations ($\tau_T = 10$).

GD values mean closeness to the Pareto front. As it can be appreciate it, values of GD, before a change in the environment occur, are always smaller due to a stability period during which particles find, and start to track the optimum. And as expected, after there is a change in the environment the GD values tend to increase. This last statement can be confirmed by looking at Figures 5.1 and 5.2.

Figure 5.1a shows the statical values obtained by using the method (a). The lines represent the average, median, worst and best GD values over 100 generations for 30 different runs; for Figure 5.1b method (b) was employed.

In table 5.3 we can appreciate the average values for (a) and (b) methods, before and after a change in the environment, for the best, worst and median values of GD for 30 runs over 100 generations. This values are graphically appreciated in figure 5.1 every 10th generation.

Figure 5.2a plots all the GD values obtained at every generation for all 30 runs using method (a), and Figure 5.2b using method (b).

In Table 5.4 we show overall statistical values of GD obtained from all 30 runs at each generation.

Experiment Set 2 (Big τ_T)

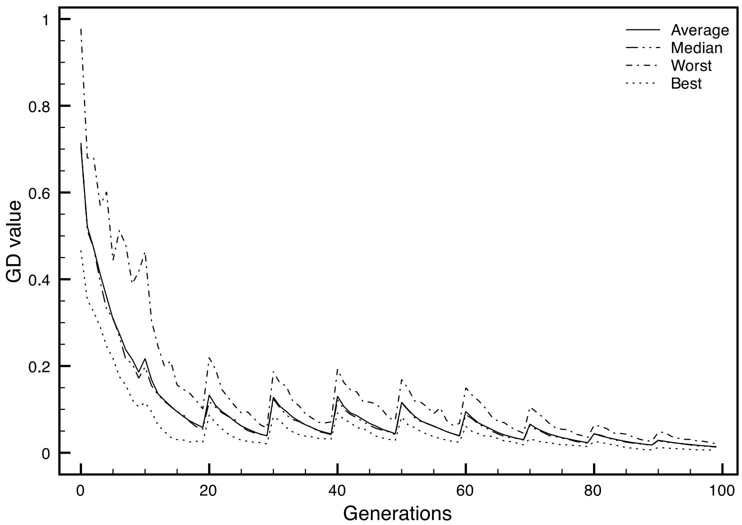
For this second set of experiments, we have a bigger lapse between environmental changes, a value for $\tau_T = 50$ will be used. Which means that to observe a total of 10 dynamic changes in the environment we will run the algorithm for 500 generations, and every 50 generations a change in the variable space will occur.

As in the previous set of experiments, Table 5.5 shows the average and standard deviations before and after a change in the variable space occurs for methods (a) and (b).

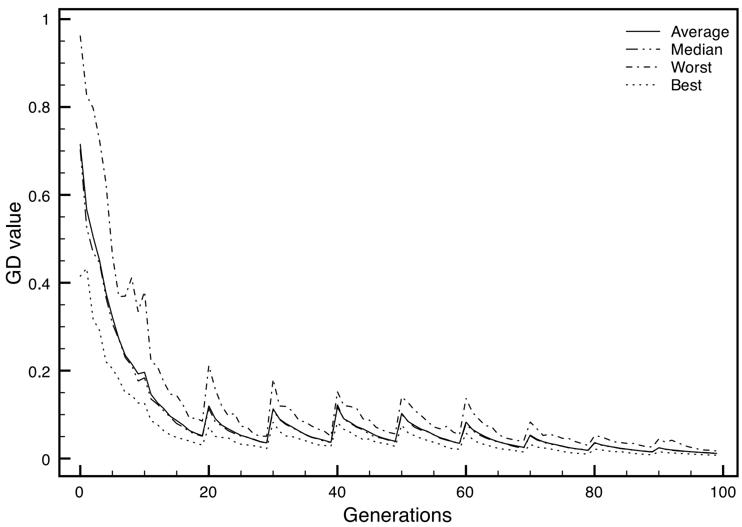
Figure 5.3a and 5.3b show the average, median, worst and best average values for all the GD values over 500 hundred generations for 30 runs.

Table 5.6 has the best, worst and median average values for method (a) and (b), before and after an environmental change.

Respectively for method (a) and (b), Figures 5.4a and 5.4b plot the GD average values for all the 30 runs over 500 generations.



(a) Graph that plots the average, median, worst and best GD metric values at each generation for 30 different runs over 100 generations.

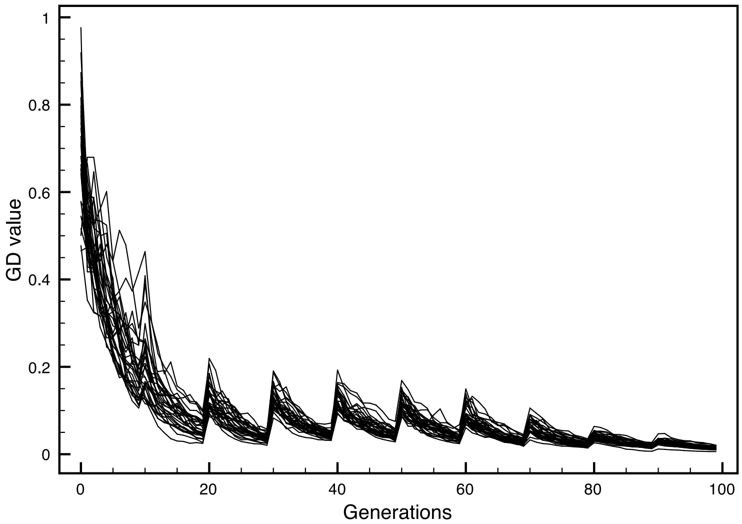


(b) Graph that plots the average, median, worst and best GD metric values at each generation for 30 different runs over 100 generations.

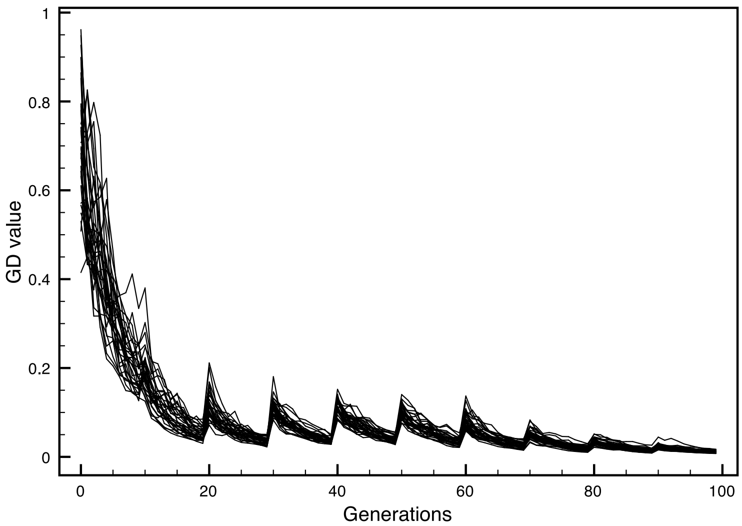
Figure 5.1: This graphical results were obtained from test function 1 (FDA1). Data in this table was obtained by performing a change in the environment every 10 generations ($\tau_T = 10$).

Environmental Change	Best			
	Before		After	
	(a)	(b)	(a)	(b)
1	0.105616	0.126458	0.465443	0.414996
2	0.024755	0.030377	0.115832	0.125412
3	0.020492	0.022309	0.086749	0.071573
4	0.031419	0.028034	0.082899	0.082845
5	0.028271	0.027554	0.092080	0.081948
6	0.024193	0.021225	0.082915	0.076188
7	0.018184	0.014889	0.060825	0.057923
8	0.014117	0.010201	0.032744	0.032261
9	0.006833	0.008406	0.026032	0.021456
10	0.006079	0.007532	0.011948	0.015602
Environmental Change	**Worst**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.416963	0.334104	0.976458	0.961780
2	0.101112	0.084992	0.463851	0.380226
3	0.057612	0.050671	0.219192	0.211538
4	0.071195	0.050572	0.190540	0.180532
5	0.075043	0.056080	0.192619	0.152161
6	0.067508	0.054177	0.168713	0.140279
7	0.045087	0.040751	0.149562	0.137002
8	0.035184	0.029669	0.105569	0.083192
9	0.027061	0.026342	0.063726	0.051714
10	0.020622	0.017165	0.046698	0.044527
Environmental Change	**Median**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.171874	0.176493	0.711042	0.702658
2	0.054719	0.050853	0.198059	0.183642
3	0.039561	0.035442	0.125305	0.114108
4	0.042889	0.036577	0.124437	0.110027
5	0.041986	0.038152	0.127511	0.122600
6	0.038639	0.034105	0.114550	0.100328
7	0.028931	0.024022	0.085958	0.078851
8	0.021986	0.018800	0.062302	0.051689
9	0.017836	0.015329	0.043310	0.035147
10	0.013382	0.011788	0.028479	0.024420

Table 5.3: This table corresponds to statistical results obtained from test function 1 (FDA1), by using a τ_T value of 10



(a) Graph showing the value of the GD metric at each generation for 30 different runs over 100 generations.



(b) Graph showing the value of the GD metric at each generation for 30 different runs over 100 generations.

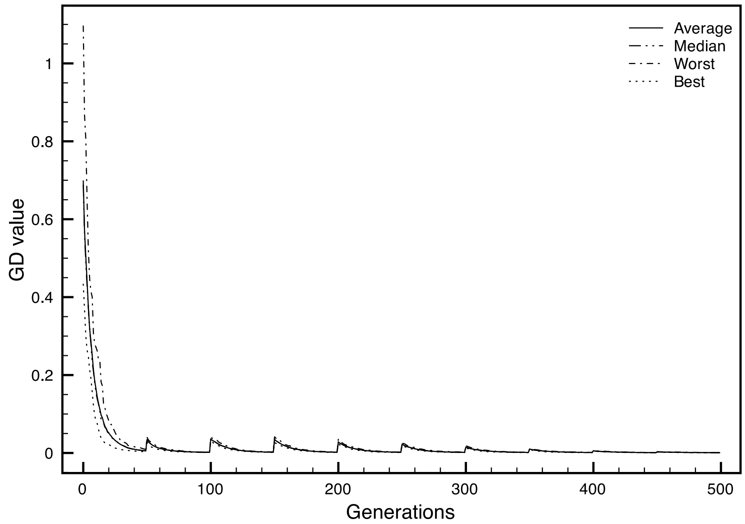
Figure 5.2: This graphical results were obtained from test function 1 (FDA1) by using a τ_T value of 10.

Statistics Set 1		
	(a)	**(b)**
Average	0.091853	0.085993
Std. Dev.	0.114679	0.120251
Best	0.006079	0.007532
Worst	0.976458	0.961780
Median	0.056322	0.048474

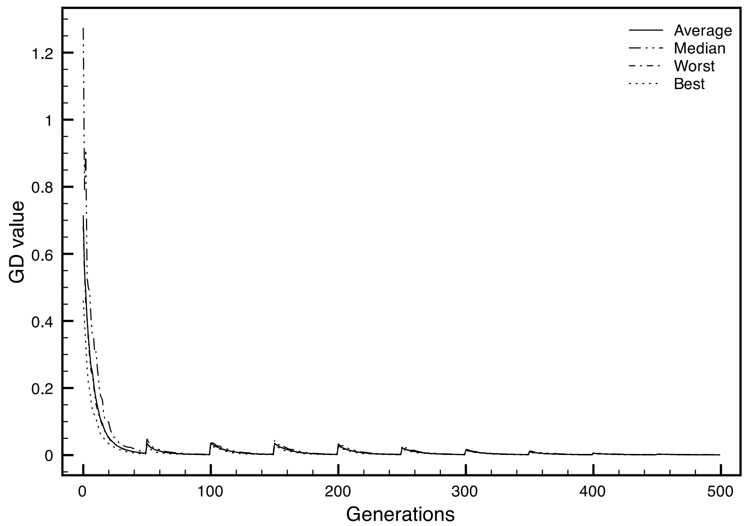
Table 5.4: From test function 1 (FDA1), statistical values obtained from one-hundred generations for 30 runs by using a τ_T value of 10.

Environmental Change	**Average**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.005441	0.005384	0.698705	0.713940
2	0.001557	0.001547	0.034201	0.034842
3	0.001934	0.001995	0.035869	0.035660
4	0.001918	0.001927	0.035010	0.035248
5	0.001754	0.001733	0.028612	0.028551
6	0.001522	0.001565	0.022633	0.022199
7	0.001324	0.001379	0.015346	0.015972
8	0.001121	0.001111	0.010100	0.010450
9	0.000831	0.000834	0.005604	0.005622
10	0.000715	0.000702	0.002723	0.002744
Environmental Change	**Standard Deviation**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.001452	0.001201	0.126801	0.159039
2	0.000175	0.000225	0.003051	0.004422
3	0.000246	0.000247	0.002189	0.002778
4	0.000239	0.000216	0.003074	0.003324
5	0.000222	0.000266	0.002630	0.002120
6	0.000254	0.000165	0.001819	0.002333
7	0.000144	0.000126	0.001542	0.001219
8	0.000122	0.000149	0.000785	0.000952
9	0.000086	0.000090	0.000420	0.000500
10	0.000089	0.000092	0.000214	0.000181

Table 5.5: This table corresponds to statistical results obtained from test function 1 (FDA1), and using a τ_T value of 50.



(a) Graph that plots the average, median, worst and best GD metric values at each generation for 30 different runs over 500 generations.

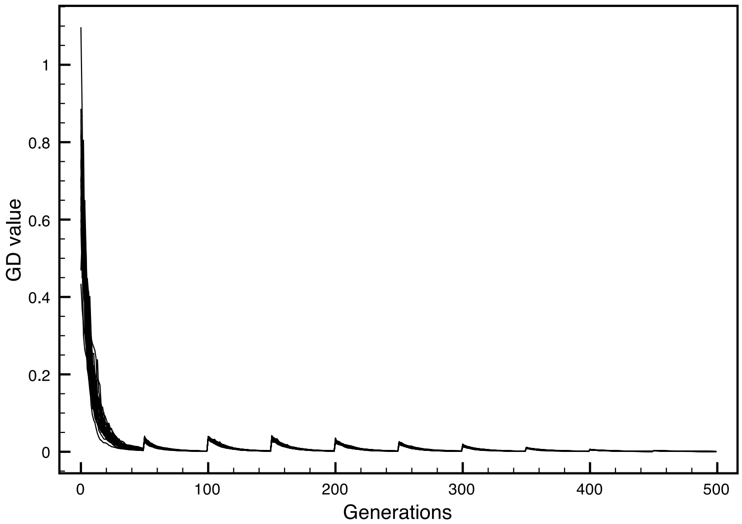


(b) Graph that plots the average, median, worst and best GD metric values at each generation for 30 different runs over 500 generations.

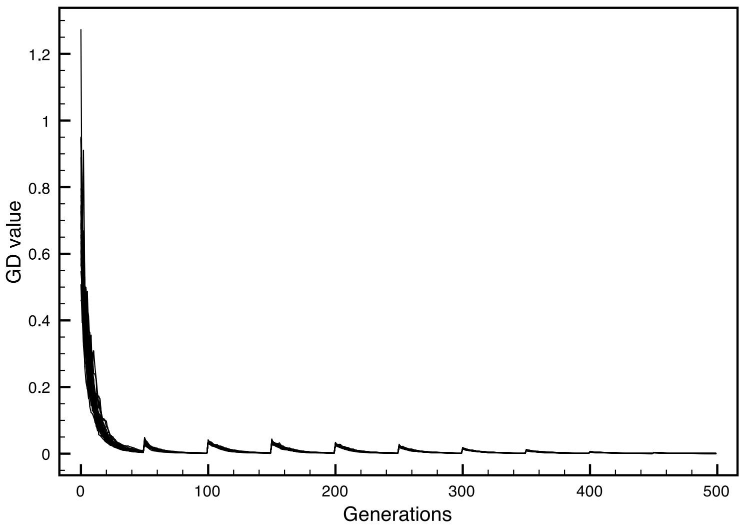
Figure 5.3: This graphical results were obtained from test function 1 (FDA1) by using a τ_T value of 50.

Environmental Change	Best			
	Before		After	
	(a)	(b)	(a)	(b)
1	0.003217	0.003487	0.433232	0.458929
2	0.001166	0.001165	0.027268	0.027402
3	0.001476	0.001384	0.031712	0.031127
4	0.001455	0.001357	0.027094	0.027155
5	0.001195	0.001220	0.023900	0.024564
6	0.001075	0.001246	0.019664	0.018889
7	0.000938	0.001151	0.012871	0.013915
8	0.000906	0.000903	0.008423	0.008362
9	0.000646	0.000680	0.004795	0.004636
10	0.000549	0.000487	0.002271	0.002343
Environmental Change	**Worst**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.009344	0.008497	1.096070	1.272900
2	0.001865	0.002109	0.040312	0.048482
3	0.002550	0.002668	0.039510	0.041074
4	0.002377	0.002402	0.041694	0.043851
5	0.002328	0.002265	0.035440	0.033997
6	0.002282	0.001964	0.026261	0.028110
7	0.001576	0.001783	0.019981	0.018344
8	0.001471	0.001524	0.011572	0.012290
9	0.001016	0.001099	0.006426	0.006794
10	0.000889	0.000896	0.003180	0.003158
Environmental Change	**Median**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.005251	0.005088	0.687735	0.682812
2	0.001593	0.001505	0.034086	0.034759
3	0.001931	0.002010	0.035879	0.035864
4	0.001874	0.001918	0.034421	0.035099
5	0.001760	0.001709	0.028077	0.028387
6	0.001512	0.001545	0.022498	0.022015
7	0.001322	0.001370	0.015023	0.015788
8	0.001084	0.001087	0.010077	0.010598
9	0.000833	0.000818	0.005589	0.005641
10	0.000716	0.000701	0.002779	0.002765

Table 5.6: This table corresponds to statistical results obtained from test function 1 (FDA1), using a value of 50 for τ_T.



(a) Graph showing the value of the GD metric at each generation for 30 different runs over 500 generations.



(b) Graph showing the value of the GD metric at each generation for 30 different runs over 500 generations.

Figure 5.4: This graphical results were obtained from test function 1 (FDA1) and using a τ_T value of 50.

Statistics Set 2		
	(a)	**(b)**
Average	0.016258	0.016163
Std. Dev.	0.060574	0.059535
Best	0.000549	0.000487
Worst	1.096070	1.272900
Median	0.003740	0.003771

Table 5.7: This table corresponds to statistical results obtained from test function 1 (FDA1), and using a value of 50 for τ_T.

Table 5.7 gives overall statistics of the two methods (a) and (b), average, standard deviation, best, worst and median for every GD value at each generation for the 30 runs.

5.4.2 Test function 2

For test function 2 (FDA2m, Equation 5.3) we have used the following parameters: $n_t = 10$; as whit test function 1, two set of experiments were performed. The first set used a $\tau_T = 10$, and the second a $\tau_T = 50$. τ_T being the number of generations for which the environment will remain stable.

Experiment Set 1 (Small τ_T)

As before, we will follow the performance of the algorithm for 100 generations during which there will occur 10 changes in the environment.

In Table 5.8 we show the average and standard deviation values for GD given by our two methods (a) and (b), before and after a change in the environment occurs. Smaller values for GD mean closeness to the Pareto front.

Figure 5.5a shows the statical values obtained by using the method (a).

Environmental Change	Average			
	Before		After	
	(a)	(b)	(a)	(b)
1	0.748916	0.674382	1.839234	1.778356
2	0.308782	0.242929	0.730043	0.660490
3	0.138341	0.101510	0.305560	0.229114
4	0.070659	0.048462	0.139524	0.099471
5	0.037009	0.024331	0.068687	0.046990
6	0.020991	0.014185	0.036225	0.023702
7	0.012770	0.008743	0.020434	0.013810
8	0.008048	0.005660	0.012508	0.008454
9	0.005397	0.003963	0.007847	0.005508
10	0.003890	0.002964	0.005246	0.003867
Environmental Change	**Standard Deviation**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.184568	0.148144	0.336436	0.231422
2	0.113780	0.064594	0.177270	0.157386
3	0.039818	0.020276	0.109495	0.057744
4	0.023573	0.012764	0.046663	0.022737
5	0.008827	0.005330	0.023186	0.014130
6	0.004664	0.003111	0.009073	0.005130
7	0.003024	0.001659	0.004218	0.002991
8	0.001958	0.001098	0.003063	0.001612
9	0.001301	0.000627	0.001910	0.001090
10	0.000746	0.000432	0.001318	0.000583

Table 5.8: This table corresponds to statistical results obtained from test function 2 (FDA2m). Data in this table was obtained by performing a change in the environment every 10 generations ($\tau_T = 10$).

The lines represent the average, median, worst and best GD values over 100 generations for 30 different runs; for Figure 5.5b method (b) was employed.

In table 5.9 we can see the average values for (a) and (b) methods, before and after a change in the environment, for the best, worst and median values of GD for 30 runs over 100 generations. This values are graphically represented in figure 5.5 every 10th generation.

Figure 5.6a plots all the GD values obtained at every generation for all 30 runs using method (a), and Figure 5.6b using method (b).

In Table 5.10 we show overall statistical values of GD obtained from all 30 runs at each generation.

Experiment Set 2 (Big τ_T)

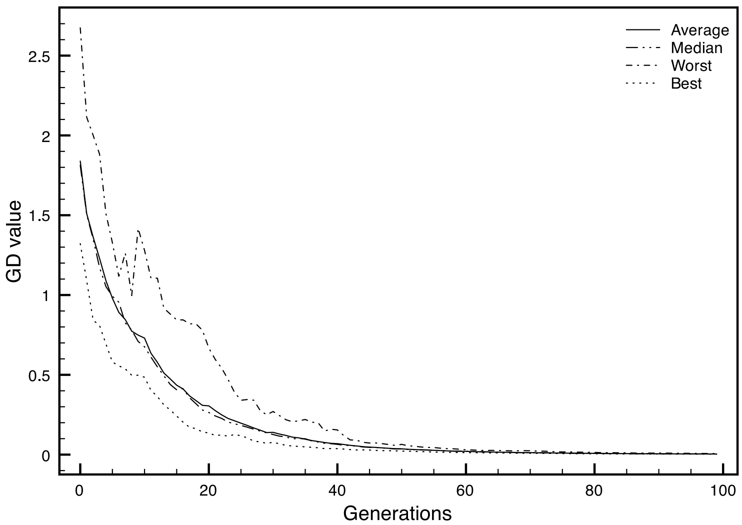
For experiment set 2, we allowed a bigger lapse between environmental changes, $\tau_T = 50$ will be used. Which means that to observe a total of 10 dynamic changes in the environment we will run the algorithm for 500 generations, and every 50 generations a change in the variable space will occur.

As in the previous set of experiments, Table 5.11 shows the average and standard deviations before and after a change in the variable space occurs for methods (a) and (b).

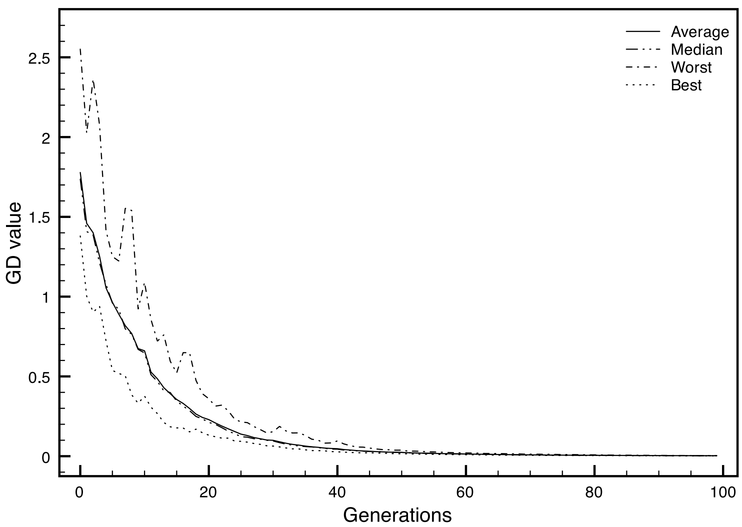
Figure 5.7a and 5.7b show the average, median, worst and best average values for all the GD values over 500 hundred generations for 30 runs.

Table 5.12 has the best, worst and median average values for method (a) and (b), before and after an environmental change.

Respectively for method (a) and (b), Figures 5.8a and 5.8b plot the GD



(a) Graph that plots the average, median, worst and best GD metric values at each generation for 30 different runs over 100 generations.

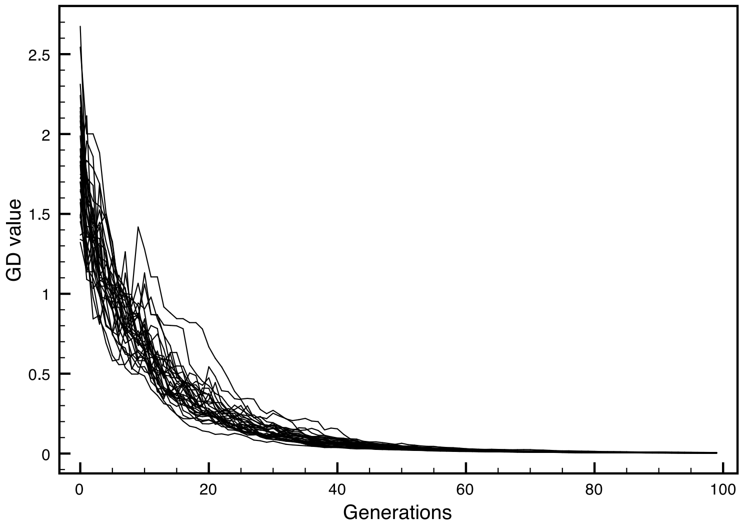


(b) Graph that plots the average, median, worst and best GD metric values at each generation for 30 different runs over 100 generations.

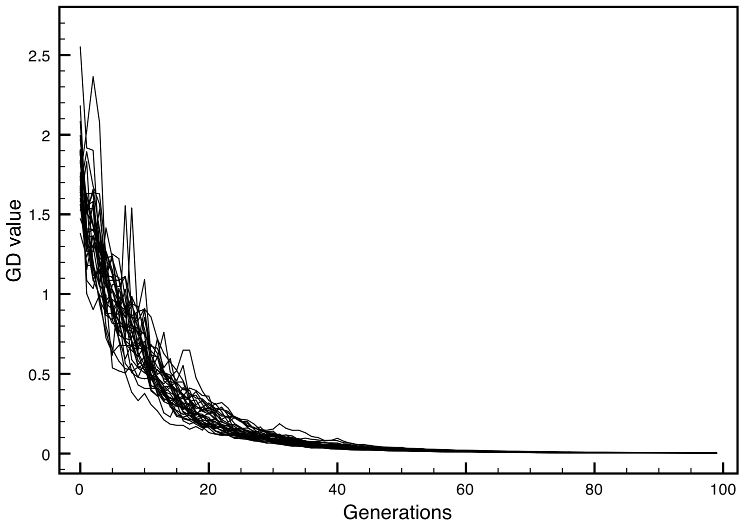
Figure 5.5: This graphical results were obtained from test function 2 (FDA2m). Data in this table was obtained by performing a change in the environment every 10 generations ($\tau_T = 10$).

Environmental Change	Best			
	Before		After	
	(a)	(b)	(a)	(b)
1	0.497418	0.331554	1.321620	1.380020
2	0.141984	0.146984	0.485066	0.377038
3	0.070547	0.065293	0.135605	0.131502
4	0.037910	0.029146	0.077350	0.063439
5	0.023635	0.015659	0.037418	0.027461
6	0.012826	0.009711	0.022038	0.015338
7	0.008363	0.005881	0.012628	0.009308
8	0.005318	0.004078	0.008562	0.005723
9	0.003651	0.002964	0.005203	0.003894
10	0.002822	0.002099	0.003630	0.002859
Environmental Change	**Worst**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	1.418160	0.921561	2.674390	2.551190
2	0.779590	0.391470	1.279890	1.090150
3	0.249716	0.146211	0.667430	0.360177
4	0.159336	0.083155	0.270285	0.153322
5	0.057669	0.037756	0.154655	0.095580
6	0.033669	0.021157	0.064216	0.037850
7	0.024694	0.013156	0.030296	0.020433
8	0.014628	0.008567	0.024438	0.012668
9	0.009521	0.005173	0.013691	0.008320
10	0.006269	0.003731	0.009841	0.004978
Environmental Change	**Median**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.709236	0.669416	1.812795	1.736775
2	0.279742	0.233836	0.675493	0.647007
3	0.135992	0.099910	0.263976	0.211549
4	0.068528	0.044567	0.125951	0.095479
5	0.035882	0.023791	0.064266	0.043915
6	0.021556	0.014328	0.035573	0.023397
7	0.012141	0.008609	0.021137	0.013444
8	0.007827	0.005562	0.012164	0.008520
9	0.005141	0.003898	0.007591	0.005429
10	0.003676	0.002926	0.004888	0.003809

Table 5.9: This table corresponds to statistical results obtained from test function 2 (FDA2m), by using a τ_T value of 10



(a) Graph showing the value of the GD metric at each generation for 30 different runs over 100 generations.



(b) Graph showing the value of the GD metric at each generation for 30 different runs over 100 generations.

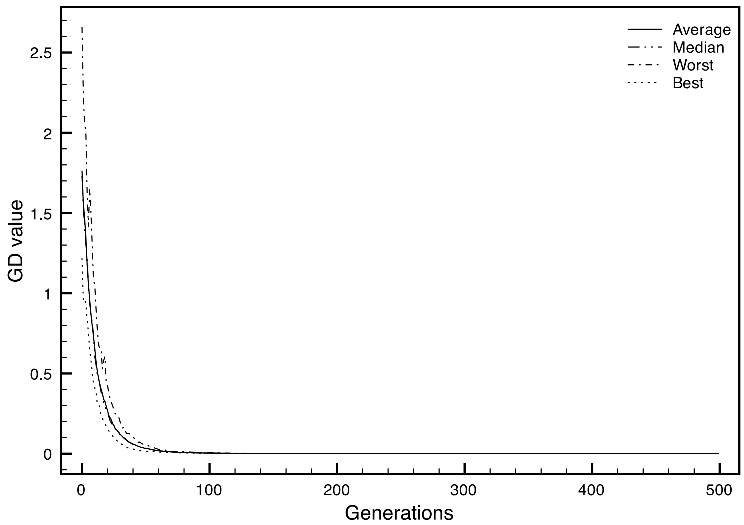
Figure 5.6: This graphical results were obtained from test function 2 (FDA2m) by using a τ_T value of 10.

Statistics Set 1		
	(a)	**(b)**
Average	0.203399	0.180628
Std. Dev.	0.368304	0.356908
Best	0.002822	0.002099
Worst	2.674390	2.551190
Median	0.035204	0.023355

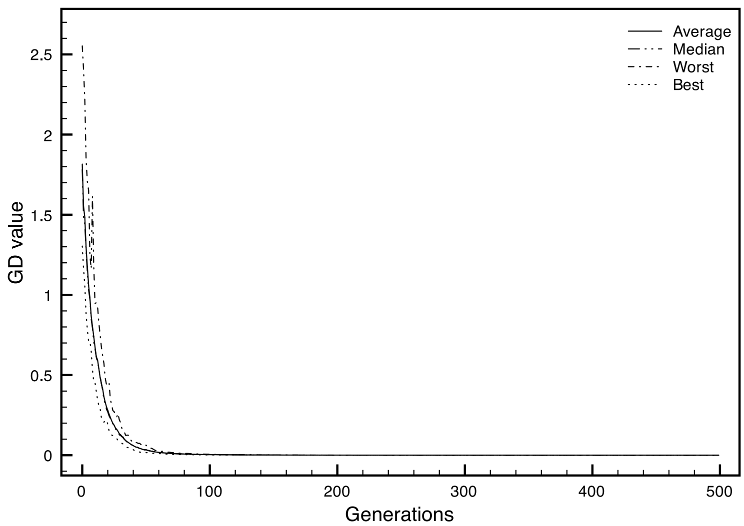
Table 5.10: From test function 2 (FDA2m), statistical values obtained from one-hundred generations for 30 runs by using a τ_T value of 10.

Environmental Change	**Average**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.033775	0.033320	1.762109	1.815938
2	0.003890	0.003763	0.034085	0.034303
3	0.001255	0.001169	0.004030	0.003896
4	0.000604	0.000563	0.001289	0.001201
5	0.000355	0.000329	0.000621	0.000578
6	0.000237	0.000222	0.000363	0.000335
7	0.000173	0.000164	0.000241	0.000225
8	0.000138	0.000130	0.000174	0.000166
9	0.000114	0.000108	0.000139	0.000131
10	0.000096	0.000092	0.000114	0.000109
Environmental Change	**Standard Deviation**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.008551	0.008869	0.305031	0.291058
2	0.000737	0.000714	0.008256	0.009691
3	0.000171	0.000172	0.000761	0.000751
4	0.000093	0.000084	0.000175	0.000168
5	0.000039	0.000044	0.000093	0.000085
6	0.000022	0.000030	0.000040	0.000046
7	0.000021	0.000014	0.000023	0.000029
8	0.000016	0.000009	0.000021	0.000015
9	0.000015	0.000008	0.000016	0.000009
10	0.000012	0.000007	0.000014	0.000010

Table 5.11: This table corresponds to statistical results obtained from test function 2 (FDA2m), and using a τ_T value of 50.



(a) Graph that plots the average, median, worst and best GD metric values at each generation for 30 different runs over 500 generations.



(b) Graph that plots the average, median, worst and best GD metric values at each generation for 30 different runs over 500 generations.

Figure 5.7: This graphical results were obtained from test function 2 (FDA2m) by using a τ_T value of 50.

Environmental Change	Best			
	Before		After	
	(a)	(b)	(a)	(b)
1	0.013953	0.014668	1.215540	1.304430
2	0.002737	0.002049	0.014985	0.016044
3	0.001027	0.000816	0.002847	0.002142
4	0.000474	0.000381	0.001036	0.000858
5	0.000289	0.000233	0.000486	0.000388
6	0.000190	0.000169	0.000300	0.000235
7	0.000138	0.000141	0.000194	0.000171
8	0.000112	0.000115	0.000139	0.000143
9	0.000086	0.000088	0.000113	0.000115
10	0.000073	0.000072	0.000086	0.000088
Environmental Change	**Worst**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.052158	0.055576	2.657170	2.552990
2	0.006011	0.005875	0.049686	0.063373
3	0.001678	0.001501	0.006097	0.006141
4	0.000825	0.000727	0.001713	0.001536
5	0.000438	0.000410	0.000831	0.000728
6	0.000294	0.000314	0.000456	0.000413
7	0.000217	0.000200	0.000300	0.000307
8	0.000182	0.000147	0.000221	0.000202
9	0.000158	0.000121	0.000184	0.000148
10	0.000127	0.000108	0.000158	0.000140
Environmental Change	**Median**			
	Before		**After**	
	(a)	**(b)**	**(a)**	**(b)**
1	0.032969	0.033524	1.728395	1.781020
2	0.003831	0.003764	0.034600	0.033345
3	0.001211	0.001161	0.003930	0.003859
4	0.000586	0.000553	0.001244	0.001174
5	0.000347	0.000324	0.000600	0.000565
6	0.000234	0.000220	0.000352	0.000330
7	0.000169	0.000163	0.000236	0.000225
8	0.000137	0.000127	0.000172	0.000164
9	0.000110	0.000109	0.000138	0.000130
10	0.000093	0.000092	0.000110	0.000109

Table 5.12: This table corresponds to statistical results obtained from test function 2 (FDA2m), using a value of 50 for τ_T.

Statistics Set 2		
	(a)	**(b)**
Average	0.040418	0.041402
Std. Dev.	0.186296	0.188429
Best	0.000073	0.000072
Worst	2.657170	2.552990
Median	0.000353	0.000329

Table 5.13: This table corresponds to statistical results obtained from test function 2 (FDA2m), and using a value of 50 for τ_T.

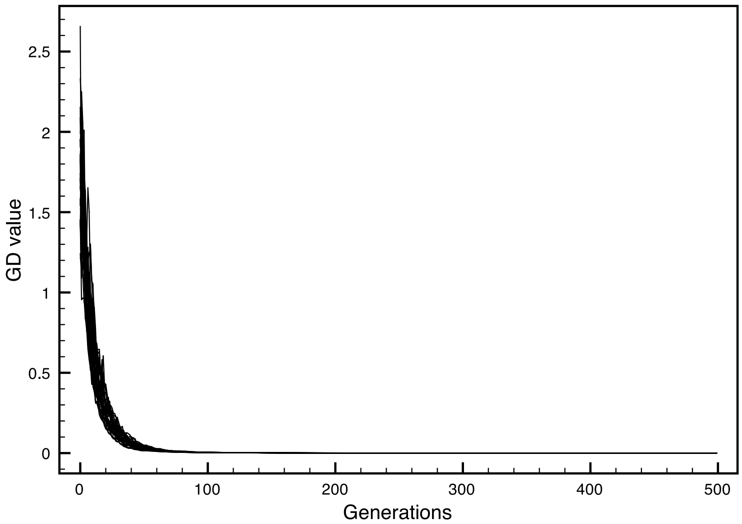
average values for all the 30 runs over 500 generations.

Table 5.13 gives overall statistics of the two methods (a) and (b), average, standard deviation, best, worst and median for every GD value at each generation for the 30 runs.

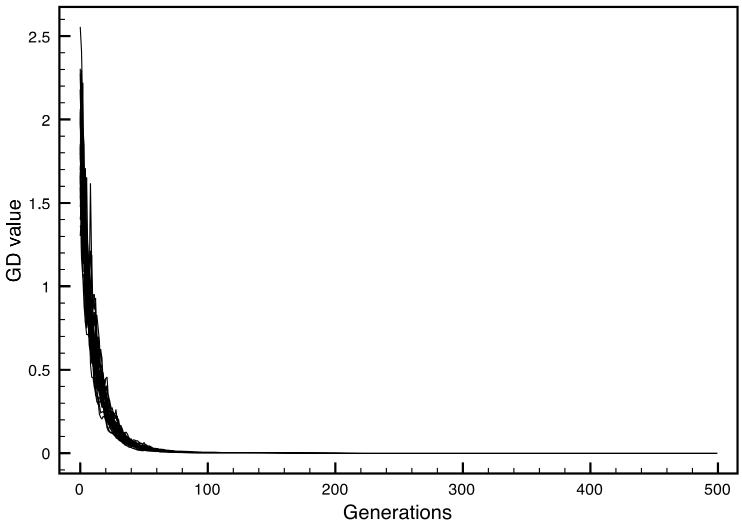
5.4.3 Discussion of Results

Let us start discussing the first test function that we have used to test the capabilities of the PSO heuristic.

First we made experiments changing the environment every 10 generations. From Table 5.2, we can observe that both methods are very similar; although it can be seen that method (b) when compared against (a), before and after the environment changes, the former is slightly better. By observing method (a) in Figure 5.1a and method (b) in Figure 5.1b, we can appreciate every 10 generations the changes on the environment and how the solutions are being affected by it; visually the similitude between both methods is not that significant, both keeping track of the optimum. In the statistics shown in Table 5.3, method (a) again is slightly outperformed by (b), which is noticed to a small degree in Figures 5.2a and 5.2b. In the overall



(a) Graph showing the value of the GD metric at each generation for 30 different runs over 500 generations.



(b) Graph showing the value of the GD metric at each generation for 30 different runs over 500 generations.

Figure 5.8: This graphical results were obtained from test function 2 (FDA2m) and using a τ_T value of 50.

performance (Table 5.4), on average method (b) shows again a slightly better performance than method (b).

Experiments performed by changing the environment every 50 generations; statistically, in Table 5.5 on averages before and after a change in the environment method (b) is again slightly better than (a), in Table 5.6, method (b) again has better statistical values than (a); and in the overall statistics (Table 5.7), method (b) also outperforms to (a) by a slight margin. Graphically both are very similar as can be seen in Figure 5.3, and 5.4.

Now let us move on to test function 2. On Table 5.8, for the first set of experiments performing 10 changes in the environment over 100 generations, we can appreciate that for this function, method (b) performs clearly better than (a); in both cases, before and after an environmental change the set of solutions found by the heuristic is always closer to the Pareto front by using method (b). This can be graphically seen, in Figure 5.5, the average GD tend to decrement faster in method (b) than in (a). And on the statistics shown in Table 5.9 it also presents better results before and after a change in the environment. When looking at the graphical representations of the average GD metric value of the 30 runs performed over 100 generations (Figure 5.6, we can see that method (b) has an slightly better performance at the start, than (a), and is clearly reflected on the overall statistics in Table 5.10.

When we perform 10 changes on the environment over 500 generations, the performance on both methods is very similar, but again a very small increment of performance on method (b) over (a) can be seen statistically, in Tables 5.11, 5.12 and 5.13; and graphically, in Figures 5.7 and 5.8.

To recapitulate we can say that for both tests functions 1 and 2, method

(b) performed slightly better than its counterpart (a), and both performed well by tracking the optimum.

The reason we believe a better performance of method (b) was achieved, is due to the fact that (b) retains the present position of the particle which makes it start anew from these position, without any previous knowledge and uninfluenced by the previous environment; whereas method (a), by retaining information, that could be inaccurate due to the change of the environment, has to recover from this misinformation and it can take it a few more generations to recover from that inaccuracy.

The reason why there appear to be smoother changes in test function 2, is because the Pareto front moves from a convex to a non-convex position and particles which are already in a convex position find it easier to move to the non-convex front, as non-dominated solutions in the convex front will be quickly dominated by those on the non-convex front.

5.5 Test and Comparison

After seeing how MOPSO-auto-*fs* performs with dynamic multi-objective functions; now, we also want to compare its performance against another algorithm. This is why we decided to compare its performance, in dynamic environments, against the technique we believe has better performance between the three (NSGA-II, PAES, MOPSO) we have compared our MOPSO-*fs* and MOPSO-auto-*fs*. From our previous experiments, we gather that NSGA-II is the one that has been consistently performing better than PAES and MOPSO; and now, we will put it to the test against MOPSO-auto-*fs* in dynamic environments.

To make this tests, NSGA-II [25] has not been modified, is the same implementation we previously used, so it might find difficulties on dealing with dynamic environments. NSGA-II used a crossover rate of 0.8, a mutation probability of $1/x$, where x is the number of variables for the given problem, and a real number representation with tournament selection; a population of 100 individuals was used.

For MOPSO-auto-*fs*, we will be using the very basic modification described in Section 5.2, and referred as method (b). We choose method (b) over method (a), given that our previous comparison and analysis between this two methods showed that method (b) performed slightly better than method (a) (see 5.4.3). Parameters for MOPSO-auto-*fs* remained the same for all the experiments here performed, these parameters are: a population of 100 particles; an external repository for 100 particles; in the velocity formula (Equation 2.12), for the acceleration coefficients c_1 and c_2 a constant value of 0.494 was used and for the inertia weight w a constant value of 0.4, both coefficients and weight values were empirically set.

Parameters needed by the functions will be stated accodingly in each test function section.

The runs we choose to plot, were selected based on the median given by one of the metrics used; e.g., out of the 30 runs for each heuristic, the one closer to the median of the GD metric was the one plotted for that particular test function.

100 Non-Dominated Solutions			
		MOPSO-*fs*	**NSGA-II**
GD	*Avg.*	0.00070159	0.087397
	S.D.	9.3197E-05	0.08093
	Med.	0.00070133	0.061802
	t-test	—	2.2523E-07 ∗
S	*Avg.*	0.004237	0.32895
	S.D.	0.00044069	0.31859
	Med.	0.0042455	0.24791
	t-test	—	6.5757E-07 ∗
D	*Avg.*	1.4173	7.7559
	S.D.	0.0016735	5.8952
	Med.	1.4168	5.9282
	t-test	—	2.0746E-07 ∗

Table 5.14: This table shows statistical results (average, standard deviation, median) of the metrics used over test function FDA1. It also shows results of a Student's *t*-test study, when comparing MOPSO-auto-*fs* against NSGA-II. [∗] Means a significant difference between results, with at least a 95% confidence level.

5.5.1 Test function FDA1

Test function proposed by Farina et al. [37] (Equation 5.2). This test problem is a type I problem (see Section 5.1), the Pareto front remains static but the variable space is the one which changes over time. For this problem we have set the value of n_t to 10, τ_T to 50, and to run for 500 generations. Which means that we will observe a total of 10 dynamic changes in the environment, during the life cycle of the algorithm, and every 50 generations is when the change in the variable space will occur, along the 500 generations executed by the algorithm. In Table 5.14 we can observe the statistical results obtained when comparing the two approaches. In Figure 5.9 we can observe a graphical comparison of the results for the two techniques.

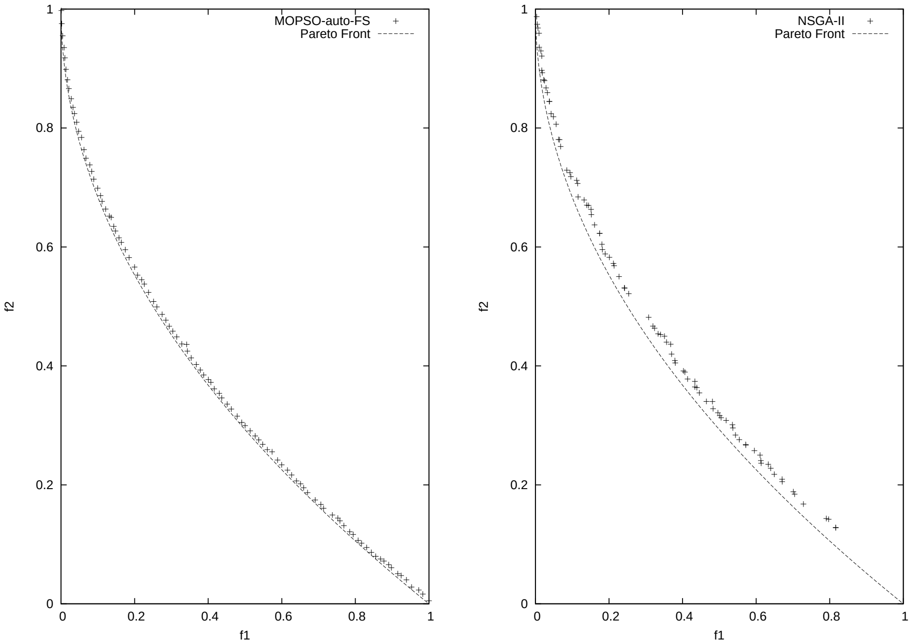


Figure 5.9: This plots correspond to the results obtained from test function FDA1.

5.5.2 Test Function FDA2m

Test function proposed by Farina et al. [37] (Equation 5.3). This test problem is a type III problem (see Section 5.1); the variable space is static, what changes over time is the Pareto front, it changes from a convex to a non-convex shape. Similar to our previous problem, we have set the value of n_t to 10, τ_T to 50, and to execute 500 generations. Which means that we will have a total of 10 dynamic changes in the environment, during the life cycle of the algorithm, and every 50 generations a change in the variable space will occur, along the 500 generations executed by the algorithm. In Table 5.15 we can observe the statistical results obtained when comparing the two approaches. In Figure 5.10 we can observe a graphical comparison of the

100 Non-Dominated Solutions			
		MOPSO-*fs*	**NSGA-II**
GD	*Avg.*	9.2118E-05	0.0074631
	S.D.	6.6822E-06	0.00046968
	Med.	9.2325E-05	0.0075635
	t-test	—	7.4993E-63 *
S	*Avg.*	0.0038377	0.0062935
	S.D.	0.00035088	0.00065415
	Med.	0.0038044	0.0062913
	t-test	—	1.4792E-25 *
D	*Avg.*	1.4136	1.4096
	S.D.	0.0017599	0.0036989
	Med.	1.4141	1.4108
	t-test	—	1.6567E-06 *

Table 5.15: This table shows statistical results (average, standard deviation, median) of the metrics used over test function FDA2m. It also shows results of a Student's *t*-test study, when comparing MOPSO-auto-*fs* against NSGA-II. [*] Means a significant difference between results, with at least a 95% confidence level.

results.

5.5.3 Discussion of Results

As can be seen from the experiments performed, comparing MOPSO-auto-*fs* against NSGA-II, our technique's results significantly outperform to those of NSGA-II. Although, we have to take into account that NSGA-II has no mechanism that helps it to know when the environment has change. Which can be a significant difference with MOPSO-auto-*fs*, who has a simple method which tells it that a change in the environment has occurred, and that it should erase its particle's memories, in order to acquire new knowledge of the environment; as described in Section 5.2, and referred as method (b).

In the first problem FDA1, we can see that statistically (Table 5.14), MOPSO-auto-*fs* outperforms to NSGA-II in all the metrics used to measure

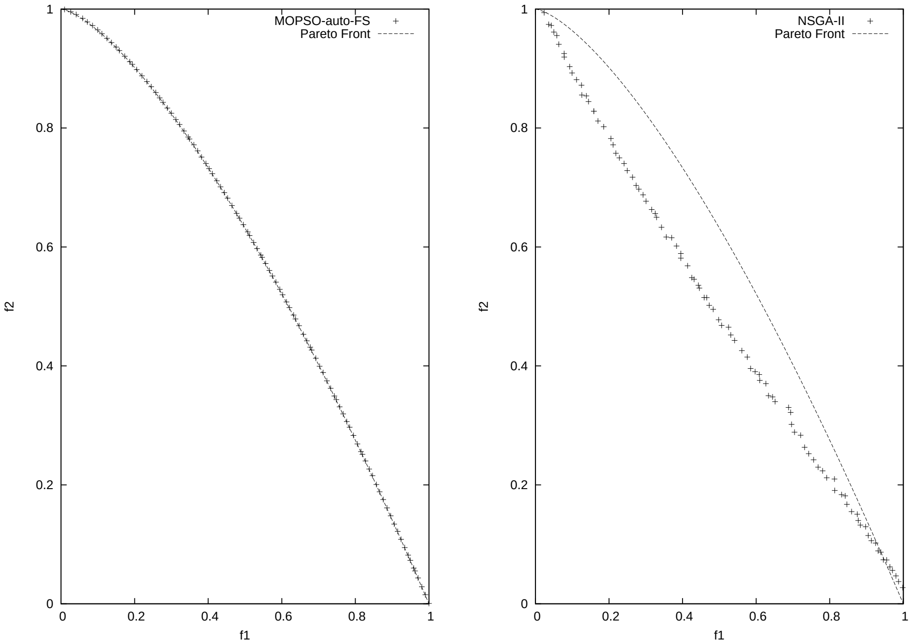


Figure 5.10: This plots correspond to the results obtained from test function FDA2.

performance on this problem. Graphically (see Figure 5.9), we can confirm this by seeing how well spread, and close to the Pareto front, are the solutions found by MOPSO-auto-*fs* in comparison to those found by NSGA-II.

For problem FDA2m, MOPSO-auto-*fs* also outperforms to NSGA-II, statistically (Table 5.15) and graphically (Figure 5.9). FDA2m being a type III problem (it has an static variable space, but a dynamic Pareto front, changing over time from a convex to a non-convex shape), we can notice in the plots, that NSGA-II is not being able to track the dynamic Pareto front shape; and the solutions it finds, almost remain in the convex shape of the initial Pareto front (see Figure 5.9).

5.6 Summary

To summarize this section, we can say that we have introduced a multi-objective PSO technique—which was developed along this thesis—into the field of multi-objective dynamic environments. And although this has not been an exhaustive study, we think we have shown that this technique is capable to track optimum within this difficult environments, and we consider it is a small step towards a promising line of investigation. More and different metrics, as well as test functions need to be used. Better mechanisms of selection after a change in the environment need to be developed. And more important, a mechanism capable to detect change in the environment is needed.

Chapter 6

Conclusion

In this section we will summarise our work and contributions presented in this thesis. We will also give an outline of research that could be derived from the work presented.

6.1 Summary

At the beginning of this thesis, we first glanced at the current state of Particle Swarm Optimisation inside the field of Multi-Objective Optimisation. After reviewing the most characteristic methods currently developed, we proceed to create a heuristic based on Particle Swarm Optimisation, capable to solve Multi-Objective Optimisation problems, with the help of fitness sharing.

The developed heuristic makes use of a fitness sharing mechanism, which helps to spread and maintain diversity in the solutions found. To show its capabilities, we tested and compared the technique against state-of-the-art heuristics specialised to solve MOPs, over different synthetic test functions taken from the specialised literature.

After testing its performance, and being aware of its potential, we then extended this heuristic by making the parameter needed by the fitness sharing mechanism, called σ_{share}, auto-adaptable. This was necessary in order to enable to our heuristic to perform without the need to empirically tune-in this parameter. We then proceeded to test and compare our heuristic, against others specialised multi-objective optimisation techniques.

From the experiments performed and showed here, in Chapters 4 and 5. We claim that MOPSO-auto-*fs*—the technique here developed—has been capable to outperform the other techniques used in this work in some circumstances, and we would strongly recommend its use over those techniques, especially under the following conditions:

- When trying to find a small number of solutions. (When the aim was to find a small number of solutions; we observed that our technique, was capable to find more diverse and closer to the Pareto front solutions, than the other heuristics.)
- When the problem involves a small number of objectives. (Our technique performed better in problems involving only two objective functions.)

Under the following conditions, we would recommend to use it, but would not guarantee that MOPSO-auto-*fs* will outperform to the other techniques by a big margin, although we believe a good diversity and better distribution of solutions could be accomplished:

- When trying to find a big number of solutions.

- When the problem involves a small number of objectives. (Our technique performed better in problems involving only two objective functions.)

Further to our conclusions, we would not recommend the use of our technique under the following circumstances:

- When the problem to optimise involves a big number of objectives.

This is because in our experiments, MOPSO-auto-*fs* did not perform well for all cases when using test functions that optimise more than two test functions; and on the other hand, the NSGA-II heuristic on average behave slightly better for those cases.

Further to this work, we have also tested the MOPSO-auto-*fs* heuristic in dynamic multi-objective environments. A couple of small modifications were made to the heuristic in order to deal with a change in the environment. This modifications were studied, and the one that presented a better performance, was compared against the NSGA-II heuristic; by using some optimisation problems from the specialised literature. The results obtained from the comparison, showed that the heuristic is capable of successfully tracking dynamic optima on a multi-objective problem. Although further work in this area, to enable to our technique to fully handle dynamic environments, clearly needs to be done. This will be further discussed in the following section.

6.2 Future Work

As mentioned in Section 2.3.1, PSO has at least two main models of connection between particles *lbest* and *gbest*, which is the way in which particles share their knowledge and experience. Within this work, we only used a *gbest* connection model. But we would like to explore the benefits or degradation in performance, by using a different connection model as the *lbest*. The main reason we suggest to do this, is that *lbest* has less susceptibility to get trapped into local optima, as it is the nature of the topology to be more diverse. Which is not the case with the *gbest* topology, that has a faster convergence rate, at the cost of less diversity. In order to accomplish this, we could propose a scheme similar to the one used in the presented work. Because we use a single repository from which we pull a *gbest* particle to guide the search; for a *lbest* topology, we could have n repositories, which will reflect specialised knowledge in different and particular areas of exploration of the Pareto front, from which we could pull the *lbest* particles to guide the search.

Although further work in this area does not have to be attached to explore only the *lbest* topology. Different social network topologies have been developed for PSO and some of this variations could benefit the performance of MOPSO-auto-*fs*.

Another area for future research is to study the behaviour of the auto-adaptive fitness sharing, specifically the σ_{share} value, and to extend the way it was implemented here. By this we mean that, from Section 4.1.1, fitness sharing has just one σ_{share} value, which might not properly space the particles in case that more than two objective functions are being evaluated, as when two objectives involved in the optimisation problem differ a lot in the scalar values they use. An alternative to this, we believe could be to use different σ_{share} values for every objective function that is being optimised.

One problem that we found in our technique, is that it does not perform well with some of the problems that involve more than two objective functions. Specifically, in problems that have hyper-planes which get smaller as they go closer to the Pareto front and when diversity needs to be maintained, in those cases convergence is not always achieved. We believe that this might be related to the balance that has to exist between proximity to the Pareto front, and diversity over the Pareto front [10]. And a closer examination to this problem is required.

In Chapter 5 we have just scratched the surface of what we are sure will be a very active line of investigation in the years to come. Dynamic multi-objective optimisation is an area which has gained attention in the last years, and techniques and heuristics representing the state-of-the-art in multi-objective optimisation will certainly be welcome into this area. From the implementation presented in this work, to solve dynamic problems, is clear that a mechanism to detect changes in the environment is the first step to make. This could be derived from the adaptation or influence of works that already deal with dynamic problems using PSO, these were briefly discussed in Seccion 2.5.

Within the general scope of this work, while performing our empirical experiments in Chapters 3, 4 and 5, we have notice a clear need for an automated setting of parameters in the PSO velocity formula (w, c_1 and

c_2). Although not an easy task, some research has been done in the area of PSO to auto-adapt these parameters [106, 107, 108], but there is little work done on the auto-adaptation of these parameters on multi-objective particle swarm optimisers [114], and we consider this to be a very interesting line of research for the future.

Bibliography

[1] Alvarez-Benitez, J. E., Everson, R. M., and Fieldsend, J. E. (2005). A MOPSO Algorithm Based Exclusively on Pareto Dominance Concepts. In *Third International Conference on Evolutionary Multi-Criterion Optimization (EMO 2005)*, pages 459–473, Guanajuato, México. Springer Berlin / Heidelberg.

[2] Angeline, P. J. (1998). Evolutionary Optimization Versus Particle Swarm Optimization: Philosophy and Performance Differences. In *Proceedings of the 7th International Conference on Evolutionary Programming VII*, volume 1447, pages 601–610, London, UK. Springer-Verlag.

[3] Ayvaz, D., Topcuoglu, H., and Gurgen, F. (2006). A Comparative Study of Evolutionary Optimization Techniques in Dynamic Environments. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'06)*, pages 1397–1398, Seattle, WA, USA.

[4] Bäck, T., Fogel, D., and Michalewicz, Z., editors (1997). *The Handbook of Evolutionary Computation*. Oxford University Press.

[5] Bartz-Beielstein, T., Limbourg, P., Mehnen, J., Schmitt, K., Parsopoulos, K. E., and Vrahatis, M. N. (2003). Particle Swarm Optimizers for Pareto

Optimization with Enhanced Archiving Techniques. In *Proceedings of the IEEE Congress on Evolutionary Computation 2003*, volume 3, pages 1780–1787.

[6] Baumgartner, U., Magele, C., and Renhart, W. (2004). Pareto Optimality and Particle Swarm Optimization. In *IEEE Transactions on Magnetics*, volume 40, pages 1172–1175.

[7] Blackwell, T. and Branke, J. (2004). Multi-Swarm Optimization in Dynamic Environments. *Lecture Notes in Computer Science*, 3005:489–500.

[8] Blackwell, T. M. and Bentley, P. J. (2002). Dynamic Search with Charged Swarms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2002)*, pages 19–26.

[9] Bleuler, S., Laumanns, M., Thiele, L., and Zitzler, E. (2003). PISA — a platform and programming language independent interface for search algorithms. In Fonseca, C. M., Fleming, P. J., Zitzler, E., Deb, K., and Thiele, L., editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Lecture Notes in Computer Science, pages 494 – 508, Berlin. Springer.

[10] Bosman, P. A. N. and Thierens, D. (2003). The Balance Between Proximity and Diversity in Multiobjective Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 7(2):174–188.

[11] Branke, J. (1999). Evolutionary Approaches to Dynamic Optimization Problems - A Survey. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'1999)*, pages 134–137.

[12] Branke, J. (2001). Evolutionary Approaches to Dynamic Optimization Problems - Updated Survey. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2001)*, pages 27–30.

[13] Carlisle, A. and Dozier, G. (2000). Adapting Particle Swarm Optimization to Dynamic Environments. In *Proceedings of the International Conference on Artificial Intelligence*, pages 429–434.

[14] Chow, C. and Tsui, H. (2004). Autonomous agent response learning by a multi-species particle swarm optimization. In *Proceedings of the Congress on Evolutionay Computation (CEC'2004)*, volume 1, pages 778–785, Portland, Oregon, USA. IEEE Service Center.

[15] Coello Coello, C. A. (1999). A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques. *Knowledge and Information Sytems. An International Journal*, 1(3):269–308.

[16] Coello Coello, C. A. (2001). List of references on evolutionary multiobjective optimization. Online. `http://delta.cs.cinvestav.mx/ ccoello/EMOO/`.

[17] Coello Coello, C. A. (2006). Evolutionary multi-objective optimization: a historical view of the field. *IEEE Computational Intelligence Magazine*, 1(1):28–36.

[18] Coello Coello, C. A. and Salazar Lechuga, M. (2002). MOPSO: A Proposal for Multiple Objective Particle Swarm Optimization. In *Proceedings of the Congress on Evolutionay Computation (CEC'02)*, volume 1, pages 1051–1056, Honolulu, HI. IEEE Press.

[19] Coello Coello, C. A., Toscano Pulido, G., and Mezura Montes, E. (2005). Current and Future Research Trends in Evolutionary Multiobjective Optimization. In Graña, M., Duro, R. J., d'Anjou, A., and Wang, P. P., editors, *Information Processing with Evolutionary Algorithms: From Industrial Applications to Academic Speculations*, pages 213–231. Springer-Verlag.

[20] Coello Coello, C. A., Toscano Pulido, G., and Salazar Lechuga, M. (2004). Handling Multiple Objectives With Particle Swarm Optimization. *IEEE Transactions on Evolutionary Computation*, 8(3):256–279.

[21] Coello Coello, C. A., Van Veldhuizen, D. A., and Lamont, G. B. (2002). *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer, New York.

[22] Dauer, J. P. and Stadler, W. (1986). Survey of Vector Optimization in Infinite-Dimensional Spaces, Part 2. *Journal of Optimization Theory and Applications*, 51(2):205–241.

[23] Deb, K. (1999). Multi-objective genetic algorithms: Problem difficulties and construction of test problems. *Evolutionary Computation*, 7(3):205–230.

[24] Deb, K. (2001). *Multi-Objective Optimization using Evolutionary Algorithms*. Wiley, Chichester.

[25] Deb, K., Agarwal, S., Pratab, A., and Meyarivan, T. (2000). A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Op-

timization: NSGA-II. In *Proceedings of the Parallel Problem Solving from Nature VI Conference*, pages 849–858. Springer.

[26] Deb, K. and Goldberg, D. E. (1989). An Investigation of Niche and Species Formation in Genetic Function Optimization. In Schaffer, J. D., editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 42–50. George Mason University, Morgan Kaufmann Publishers.

[27] Deb, K., Pratab, A., Agarwal, S., and Meyarivan, T. (2002a). A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197.

[28] Deb, K., Thiele, L., Laumanns, M., and Zitzler, E. (2001). Scalable test problems for evolutionary multi-objective optimization. Technical Report 112, Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich, Zürich, Switzerland.

[29] Deb, K., Thiele, L., Laumanns, M., and Zitzler, E. (2002b). Scalable multiobjective optimization test problems. In *Congress on Evolutionary Computation (CEC'2002)*, volume I, pages 825–830, Piscataway, New Jersey. IEEE Service Center.

[30] Eberhart, E. C. and Shi, Y. (2001a). Tracking and Optimizing Dynamic Systems with Particle Swarms. In *Proceedings of the 2001 Congress on Evolutionary Computation*, pages 94–100. IEEE.

[31] Eberhart, R. C. and Shi, Y. (1998). Comparison between Genetic Algorithms and Particle Swarm Optimization. In *Proceedings of the 7th In-*

ternational Conference on Evolutionary Programming VII, volume 1447, pages 611–616, London, UK. Springer-Verlag.

[32] Eberhart, R. C. and Shi, Y. (2001b). Particle Swarm Optimization: Development Application and Resources. In *Proceedings of the 2001 Congress on Evolutionary Computation*, pages 81–86. IEEE.

[33] Edgeworth, F. Y. (1881). *Mathematical Physics*. P. Keagan, London, England.

[34] Engelbrecht, A. P. (2005). *Fundamentals of Computational Swarm Intelligence*. John Wiley and Sons Ltd.

[35] Engelbrecht, A. P. (2006). Particle Swarm Optimization: Where Does It Belong? In *Proceedings of the IEEE Swarm Intelligence Symposium 2006*, pages 48–54, Indianapolis, Indiana.

[36] Farina, M., Deb, K., and Amato, P. (2003). Dynamic Multiobjective Optimization Problems: Test Cases, Approximations, and Applications. In *Proceedings of the Second International Conference on Evolutionary Multi-Criterion Optimization (EMO'03)*, volume 2632 of *Lecture Notes in Computer Science*, pages 311–326, Faro, Portugal. Springer-Verlag Berlin Heidelberg.

[37] Farina, M., Deb, K., and Amato, P. (2004). Dynamic Multiobjective Optimization Problems: Test Cases, Approximations, and Applications. *IEEE Transactions on Evolutionary Computation*, 8(5):425–442.

[38] Fieldsend, J. and Singh, S. (2002). A Multi-Objective Algorithm based

upon Particle Swarm Optimisation, an Efficient Data Structure and Turbulence. In *Proceedings of UK Workshop on Computational Intelligence (UKCI'02)*, pages 37–44, Birmingham, UK.

[39] Fieldsend, J. E. (2004). Multi-objective particle swarm optimisation methods. Technical Report 419, Department of Computer Science, University of Exeter.

[40] Fieldsend, J. E., Everson, R. M., and Singh, S. (2003). Using Uncostrained Elite Archives for Multiobjetive Optimization. *IEEE Transactions on Evolutionary Computation*, 7(3):305–323.

[41] Fogel, L. J. (1999). *Intelligence Through Simulated Evolution: Forty Years of Evolutionary Programming.* John Wiley and Sons.

[42] Fonseca, C. M. and Fleming, P. J. (1993). Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization. In Forrest, S., editor, *Genetic Algorithms: Proceedings of the Fifth International Conference*, pages 416–423. Morgan Kaufmann.

[43] Fonseca, C. M. and Fleming, P. J. (1995). Multiobjective genetic algorithms made easy: selection sharing and mating restriction. In *Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, Sheffield, UK.

[44] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.

[45] Goldberg, D. E. and Richardson, J. (1987). Genetic Algorithms with

Sharing for Multimodal Function Optimization. In Grefenstette, J., editor, *Proceedings of the 2nd International Conference on Genetic Algorithms*, pages 41–49, Hillsdale, New Jersey. Lawrence Erlbaum Associates.

[46] Gong, D.-W., Zhang, Y., and Zhang, J.-H. (2005). Multi-objective Particle Swarm Optimization Based on Minimal Particle Angle. In *Proceedings of Advances in Intelligent Computing: International Conference on Intelligent Computing (ICIC 2005), Part I*, volume 3644 of *Lecture Notes in Computer Science*, pages 571–580, Hefei, China. Springer Berlin / Heidelberg.

[47] Hansen, M. P. and Jaszkiewicz, A. (1998). Evaluating the quality of approximations to the non-dominated set. Technical Report IMM-REP-1998-7, Technical University of Denmark.

[48] Heitkötter, J. and Beasley, D., editors (2000). *The Hitch-Hiker's Guide to Evolutionary Computation: A list of Frequently Asked Questions (FAQ).* Available via anonymous FTP from ftp://rtfm.mit.edu/pub/usenet/news.answers/ai-faq/genetic/.

[49] Ho, S., Yang, S., Ni, G., Lo, E. W., and Wong, H. C. (2005). A particle swarm optimization-based method for multiobjective design opti- mizations. In *IEEE Transactions on Magnetics*, volume 41, pages 1756–1759.

[50] Ho, S.-J., Ku, W.-Y., Jou, J.-W., Hung, M.-H., and Ho, S.-Y. (2006). Intelligent Particle Swarm Optimization in Multi-objective Problems. In *Proceedings of Advances in Knowledge Discovery and Data Mining: 10th Pacific-Asia Conference (PAKDD 2006)*, volume 3918 of *Lecture Notes in*

Computer Science, pages 790–800, Singapore. Springer Berlin / Heidelberg.

[51] Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan.

[52] Horn, J. (1997). Multicriterion decision making. In Bäck, T., Fogel, D., and Michalewicz, Z., editors, *The Handbook of Evolutionary Computation*, pages F1.9:1–F1.9:15. Oxford University Press.

[53] Horn, J., Nafpliotis, N., and Goldberg, D. E. (1994). A Niched Pareto Genetic Algorithm for Multiobjective Optimization. In *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, volume 1, pages 82–87, Piscataway, New Jersey. IEEE Service Center.

[54] Hu, X. and Eberhart, R. C. (2002). Multiobjective Optimization Using Dynamic Neighborhood Particle Swarm Optimization. In *Proceedings of the Congress on Evolutionay Computation (CEC'02)*, volume 2, pages 1677–1681, Honolulu, HI. IEEE Press.

[55] Hu, X., Eberhart, R. C., and Shi, Y. (2003). Particle Swarm with Extended Memory for Multiobjective Optimization. In *Proceedings of the IEEE Swarm Intelligence Symposium 2003 (SIS 2003)*, pages 193–197, Indianapolis, Indiana. IEEE.

[56] Hu, X., Shi, Y., and Eberhart, R. C. (2004). Recent advances in particle swarm. In *Proceedings of the IEEE Congress on Evolutionary Computation 2004*. IEEE.

[57] Huang, V., Suganthan, P., and Liang, J. (2006). Comprehensive learning particle swarm optimizer for solving multiobjective optimization problems. *International Journal of Intelligent Systems*, 21(2):209–226.

[58] Hurwicz, L. (1958). Programming in Linear Spaces. In Arrow, K. J., Hurwicz, L., and Uzawa, H., editors, *Studies in Linear and Nonlinear Programming*, chapter 4, pages 38–102. Stanford University Press, Stanford, California.

[59] Janson, S. and Merkle, D. (2005). A New multi-objective Particle Swarm Optimization Algorithm Using Clustering Applied to Automated Docking. In *Proceedings of the Hybrid Metaheuristics: Second International Workshop*, volume 3636 of *Lecture Notes in Computer Science*, pages 128–141, Barcelona, Spain. Springer Berlin / Heidelberg.

[60] Kennedy, J. and Eberhart, R. C. (1995). Particle Swarm Optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, Piscataway, New Jersey. IEEE Service Center.

[61] Kennedy, J. and Eberhart, R. C. (2001). *Swarm Intelligence*. Morgan Kaufmann, San Francisco, California.

[62] Knowles, J. and Corne, D. (1999). The Pareto Archived Evolution Strategy: A New Baseline Algorithm for Pareto Multiobjective Optimisation. In *Proceedings of the Congress on Evolutionary Computation*, volume 1, pages 98–105, Mayflower Hotel, Washington D.C., USA. IEEE Press.

[63] Knowles, J. D. and Corne, D. W. (2000). Approximating the Nondomi-

nated Front Using the Pareto Archived Evolution Strategy. *Evolutionary Computation*, 8(2):149–172.

[64] Knowles, J. D. and Corne, D. W. (2002). On metrics for comparing non-dominated sets. In *Proceedings of the 2002 Congress on Evolutionary Computation Conference (CEC '02)*, pages 711–716. IEEE Press.

[65] Knowles, J. D., Thiele, L., and Zitzler, E. (2006). A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers. TIK-Report 214, ETH Zürich.

[66] Koopmans, T. C. (1951). Analysis of Production as an Efficient Combination of Activities. In Koopmans, T. C., editor, *Activity Analysis of Production and Allocation*, Cowles Commission Monograph No. 13, chapter 3, pages 33–97. John Wiley and Sons, New York.

[67] Koza, J. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, Cambridge, Massachusetts.

[68] Krami, N., El-Sharkawi, M. A., and Akherraz, M. (2006). Multi Objective Particle Swarm Optimization Technique for Reactive Power Planning. In *Proceedings of the IEEE Swarm Intelligence Symposium 2006*, pages 170–174, Indianapolis, Indiana, USA. IEEE Press.

[69] Kuhn, H. W. and Tucker, A. W. (1951). Nonlinear Programming. In Neyman, J., editor, *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492, Berkeley, California.

[70] Kursawe, F. (1991). A variant of evolution strategies for vector optimization. In *PPSN I: Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, pages 193–197, London, UK. Springer-Verlag.

[71] Li, T., Wei, C., and Pei, W. (2003). PSO with sharing for multimodal function optimization. In *Proceedings of the 2003 International Conference on Neural Networks and Signal Processing*, volume 1, pages 450–453.

[72] Li, X. (2003). A Non-dominated Sorting Particle Swarm Optimizer for Multiobjective Optimization. In *GECCO*, volume 2723 of *Lecture Notes in Computer Science*, pages 37–48. Springer-Verlag.

[73] Li, X. (2004). Better Spread and Convergence: Particle Swarm Multiobjective Optimization Using the Maximin Fitness Function. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2004)*, pages 117–128, Seattle, WA, USA. Springer Berlin / Heidelberg.

[74] Liang, J. J., Qin, A. K., Suganthan, P. N., and Baskar, S. (2004). Evaluation of Comprehensive Learning Particle Swarm Optimizer. In *Proceedings of the Neural Information Processing: 11th International Conference (ICONIP 2004)*, volume 3316 of *Lecture Notes in Computer Science*, pages 230–235, Calcutta, India. Springer Berlin / Heidelberg.

[75] Little, I. M. D. (1950). *A Critique of Welfare Economics*. Oxford University Press.

[76] Lu, H. (2003). Dynamic Population Strategy Assisted Particle Swarm Optimization in Multiobjective Evolutionary Algorithm Design. IEEE

Neural Network Society, IEEE NNS Student Research Grants 2002 - Final Reports 2003.

[77] Lu, H. and Yen, G. G. (2002). Dynamic population size in multiobjective evolutionary algorithm. In *Proceedings of the Congress on Evolutionay Computation (CEC'02)*, volume 2, pages 1648–1653, Honolulu, HI, USA.

[78] Mahfoud, S. W. (1995). *Niching Methods for Genetic Algorithms.* PhD thesis, Illinois Genetic Algorithms Laboratory (IlliGAL), Department of General Engineering, University of Illinois at Urbana-Champaign.

[79] Mahfouf, M., Chen, M., and Linkens, D. A. (2004). Adaptive Weighted Particle Swarm Optimisation for Multi-objective Optimal Design of Alloy Steels. In *Parallel Problem Solving from Nature - PPSN VIII: 8th International Conference*, volume 3242, pages 762–771, Birmingham, UK. Springer Berlin / Heidelberg.

[80] Meng, H., Zhang, X., and Liu, S. (2005a). A Co-evolutionary Particle Swarm Optimization-Based Method for Multiobjective Optimization. In *Proceedings of AI 2005: Advances in Artificial Intelligence: 18th Australian Joint Conference on Artificial Intelligence*, volume 3809 of *Lecture Notes in Computer Science*, pages 349–359, Sydney, Australia. Springer Berlin / Heidelberg.

[81] Meng, H., Zhang, X., and Liu, S. (2005b). Intelligent Multiobjective Particle Swarm Optimization Based on AER Model. In *Proceedings of Progress in Artificial Intelligence: 12th Portuguese Conference on Artifi-*

cial Intelligence (EPIA 2005), volume 3808 of *Lecture Notes in Computer Science*, pages 178–189, Covilha, Portugal. Springer Berlin / Heidelberg.

[82] Michalewicz, Z. and Fogel, D. B. (2000). *How to Solve It: Modern Heuristics*. Springer-Verlag, Berlin.

[83] Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. The MIT Press, Cambridge, Massachusetts.

[84] Moore, J. and Chapman, R. (1999). Application of particle swarm to multiobjective optimization. Department of Computer Science and Software Engineering, Auburn University. (Unpublished).

[85] Mostaghim, S. and Teich, J. (2003a). The Role of ϵ-dominance in Multi Objective Particle Swarm Optimization Methods. In *Proceedings of the IEEE Congress on Evolutionary Computation 2003*, pages 1764–1771.

[86] Mostaghim, S. and Teich, J. (2003b). Strategies for Finding Good Local Guides in Multi-objective Particle Swarm Optimization. In *Proceedings of the IEEE Swarm Intelligence Symposium 2003 (SIS 2003)*, pages 26–33, Indianapolis, Indiana. IEEE.

[87] Mostaghim, S. and Teich, J. (2004). Covering Pareto-optimal Fronts by Subswarms in Multi-objective Particle Swarm Optimization. In *Proceedings of the IEEE Congress on Evolutionary Computation 2004*, pages 1404–1411.

[88] Osyczka, A. (2002). *Evolutionary Algorithms for Single and Multicrite-*

ria Design Optimization, volume 79 of *Studies in fuzzines and soft computing*. Physica-Verlag, Heidelberg, Germany.

[89] Pareto, V. F. D. (1906). *Manuale di Economia Politica*. Societa Editrice Libraria, Milan, Italy.

[90] Parsopoulos, K. E., Tasoulis, D. K., and Vrahatis, M. N. (2004). Multiobjective Optimization Using Parallel Vector Evaluated Particle Swarm Optmization. In *Proceedings of the IASTED International Conference on Artificial Intelligence and Applications (AIA 2004)*, volume 2, pages 823–828, Innsbruck, Austria. ACTA Press.

[91] Parsopoulos, K. E. and Vrahatis, M. N. (2002a). Particle Swarm Optimization Method in Multiobjective Problems. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 603–607, Madrid, Spain. ACM Press.

[92] Parsopoulos, K. E. and Vrahatis, M. N. (2002b). Recent Approaches to Global Optimization Problems Through Particle Swarm Optimization. *Natural computing*, 1(2–3):235–306.

[93] Raquel, C. R. and Naval, J. P. C. (2005). An Effective Use of Crowding Distance in Multiobjective Particle Swarm Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2005)*, pages 257–264, Washington DC, USA. ACM Press.

[94] Ray, T. and Liew, K. M. (2002). A Swarm Metaphor for Multiobjective Design Optimization. *Engineering Optimization*, 34(2):141–153.

[95] Reeves, C. R. and Rowe, J. E. (2002). *Genetic Algorithms - Principles and Perspectives: A Guide to GA Theory*. Kluwer, Boston.

[96] Reyes-Sierra, M. and Coello Coello, C. A. (2005a). Fitness Inheritance in Multi-Objective Particle Swarm Optimization. In *Proceedings of the IEEE Swarm Intelligence Symposium 2005*, pages 116–123, Pasadena, California, USA. IEEE Press.

[97] Reyes-Sierra, M. and Coello Coello, C. A. (2005b). Improving PSO-based Multi-Objective Optimization using Crowding, Mutation and ϵ-Dominance. In *Third International Conference on Evolutionary Multi-Criterion Optimization (EMO 2005)*, pages 505–519, Guanajuato, México. Springer Berlin / Heidelberg.

[98] Reyes-Sierra, M. and Coello Coello, C. A. (2005c). A Study of Fitness Inheritance and Approximation Techniques for Multi-Objective Particle Swarm Optimization. In *Proceedings of the 2005 IEEE Congress on Evolutionay Computation (CEC 2005)*, volume 1, pages 65–72, Edinburgh, Scotland, UK. IEEE Service Center.

[99] Reyes-Sierra, M. and Coello Coello, C. A. (2006a). Multi-Objective Particle Swarm Optimizers: A Survey of the State-of-the-Art. Technical Report EVOCINV-01-2006, Evolutionary Computation Group at CINVESTAV, Sección de Computación, Departamento de Ingeniería Eléctrica, CINVESTAV-IPN, México.

[100] Reyes-Sierra, M. and Coello Coello, C. A. (2006b). On-line Adaptation in Multi-Objective Particle Swarm Optimization. In *Proceedings of*

the IEEE Swarm Intelligence Symposium 2006, pages 61–68, Indianapolis, Indiana, USA.

[101] Salazar Lechuga, M. and Rowe, J. E. (2005). Particle Swarm Optimization and Fitness Sharing to solve Multi-Objective Optimization Problems. In *Proceedings of the 2005 IEEE Congress on Evolutionay Computation (CEC 2005)*, volume 2, pages 1204–1211, Edinburgh, Scotland, UK. IEEE Service Center.

[102] Salazar Lechuga, M. and Rowe, J. E. (2006). Particle Swarm Optimization and auto-Fitness Sharing to solve Multi-Objective Optimization Problems. In *Proceedings of the IEEE Swarm Intelligence Symposium 2006*, pages 90–97, Indianapolis, Indiana, USA.

[103] Sareni, B. and Krähenbühl, L. (1998). Fitness Sharing and Niching Methods Revisited. *IEEE Transactions on Evolutionary Computation*, 2(3):97–106.

[104] Schaffer, J. D. (1985). Multiple Objective Optimization with Vector Evaluated Genetic Algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 93–100, Mahwah, NJ, USA. Lawrence Erlbaum Associates, Inc.

[105] Schott, J. R. (1995). Fault tolerant design using single and multicriteria genetic algorithm optimization. Master's thesis, Massachusetts Institute of Technology.

[106] Shi, Y. and Eberhart, R. C. (1998a). A Modified Particle Swarm Op-

timizer. In *IEEE International Conference of Evolutionary Computation*, pages 69—73, Alaska. IEEE Press.

[107] Shi, Y. and Eberhart, R. C. (1998b). Parameter Selection in Particle Swarm Optimization. In *Evolutionary Programming VII: Proceedings of EP 98*, pages 591–600.

[108] Shi, Y. and Eberhart, R. C. (1999). Empirical Study of Particle Swarm Optimization. In *Proceedings of the Congress on Evolutionary Computation*, pages 1945–1949, Washington D.C., USA. IEEE Service Center, Piscataway, NJ.

[109] Song, M.-P. and Gu, G.-C. (2004). Research on Particle Swarm: a Review. In *Proceedings of 2004 International Conference on Machine Learning and Cybernetics*, volume 4, pages 2236–2241. IEEE.

[110] Srinivas, N. and Deb, K. (1994). Multiobjective Optimization Using Nondominated Sorting in Genetic Algorithms. *Evolutionary Computation*, 2(3):221–248.

[111] Srinivasan, D. and Seow, T. H. (2003). Particle Swarm Inspired Evolutionary Algorithm (PS-EA) for Multiobjective Optimization Problem. In *Proceedings of the IEEE Congress on Evolutionary Computation 2003 (CEC'2003)*, volume 4, pages 2292–2297, Canberra, Australia. IEEE Press.

[112] Stadler, W. (1979). A Survey of Multicriteria Optimization or the Vector Maximum Problem, Part I: 1776–1960. *Journal of Optimization Theory and Applications*, 29(4):1–52.

[113] Stadler, W. (1986). Initiators of multicriteria optimization. In Jahn, J. and Krabs, W., editors, *Recent Advances and Historical Development of Vector Optimization*, pages 3–47. Springer-Verlag, Berlin.

[114] Toscano Pulido, G. (2005). *On the Use of Self-Adaptation and Elitism for Multiobjective Particle Swarm Optimization.* PhD thesis, Centro de Investigación y Estudios Avanzados del Instituto Politécnico Nacional.

[115] Toscano Pulido, G. and Coello Coello, C. A. (2004). Using Clustering Techniques to Improve the Performance of a Multi-Objective Particle Swarm Optimizer. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2004)*, volume 3102, pages 225–237, Seattle, WA, USA. Springer Berlin / Heidelberg.

[116] Triantaphyllou, E. (2000). *Multi-Criteria Decision Making Methods: A Comparative Study*, volume 44 of *Applied Optimization.* Kluwer Academic Publishers, Dordretcht, The Netherlands.

[117] van den Bergh, F. (2002). *An Analysis of Particle Swarm Optimizers.* PhD thesis, University of Pretoria.

[118] Van Veldhuizen, D. A. (1999). *Multiobjective Evolutionary Algorithms: Classifications, Analyses and New Innovations.* PhD thesis, Air Force Institute of Technology Air University.

[119] Van Veldhuizen, D. A. and Lamont, G. B. (1998a). Evolutionary Computation and Convergence to a Pareto Front. In Koza, J. R., editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, pages 221–

228, University of Wisconsin, Madison, Wisconsin, USA. Stanford University Bookstore.

[120] Van Veldhuizen, D. A. and Lamont, G. B. (1998b). Multiobjective Evolutionary Algorithm Research: A History and Analysis. Technical Report TR-98-03, Department of Electrical and Computer Engineering, Graduate School of Engineering, Air Force Institute of Technology , Wright-Patterson AFB, Ohio.

[121] Van Veldhuizen, D. A. and Lamont, G. B. (2000). Multiobjective Evolutionary Algorithms: Analyzing the State-of-the-Art. *Evolutionary Computation*, 8(2):125–147.

[122] Vesterstrøm, J. and Thomsen, R. (2004). A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In *Proceedings of the Congress on Evolutionay Computation (CEC'2004)*, volume 2, pages 1980–1987.

[123] Villalobos-Arias, M. A., Toscano Pulido, G., and Coello Coello, C. A. (2005). A Proposal to use Stripes to Maintain Diversity in a Multi-Objective Particle Swarm Optimizer. In *Proceedings of the IEEE Swarm Intelligence Symposium 2005*, pages 22–29, Pasadena, California, USA. IEEE Press.

[124] Wang, L. and Singh, C. (2006). Multi-Objective Stochastic Power Dispatch Through A Modified Particle Swarm Optimization Algorithm. In *Proceedings of the IEEE Swarm Intelligence Symposium 2006*, pages 128–135, Indianapolis, Indiana, USA. IEEE Press.

[125] Zhang, L., Yu, H., and Hu, S. (2003a). A New Approach to Improve Particle Swarm Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2003)*, volume 2723, pages 134–139. Springer Berlin / Heidelberg.

[126] Zhang, L., Zhou, C., Liu, X., Ma, Z., Ma, M., and Liang, Y. (2003b). Solving Multi Objective Optimization Problems Using Particle Swarm Optimization. In *Proceedings of the Congress on Evolutionay Computation (CEC'2003)*, volume 4, pages 2400–2405, Canberra, Australia. IEEE Press.

[127] Zhang, X., Meng, H., and Jiao, L. (2005). Intelligent Particle Swarm Optimization in Multiobjective Optimization. In *Proceedings of the 2005 IEEE Congress on Evolutionay Computation (CEC 2005)*, pages 714–719, Edinburgh, Scotland, UK. IEEE Press.

[128] Zhao, B. and Cao, Y. (2005). Multiple objective particle swarm optimization technique for economic load dispatch. *Journal of Zhejiang University SCIENCE*, 6A(5):420–427.

[129] Zitzler, E., Deb, K., and Thiele, L. (1999). Comparison of Multiobjective Evolutionary Algorithms: Empirical Results (Revised Version). Technical report, Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich, Switzerland.

[130] Zitzler, E., Deb, K., and Thiele, L. (2000). Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary Computation*, 8(2):173–195.

[131] Zitzler, E., Laumanns, M., and Bleuler, S. (2004). A Tutorial on Evolutionary Multiobjective Optimization. In *Metaheuristics for Multiobjective Optimisation*, volume 535 of *Lecture Notes in Economics and Mathematical Systems*, pages 3–37. Springer.

[132] Zitzler, E., Laumanns, M., and Thiele, L. (2001). SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Zürich, Switzerland.

[133] Zitzler, E. and Thiele, L. (1998a). An Evolutionary Algorithm for Multiobjective Optimization: The Strength Pareto Approach. Technical Report 43, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Zürich, Switzerland.

[134] Zitzler, E. and Thiele, L. (1998b). Multiobjective optimization using evolutionary algorithms - a comparative case study. In *Parallel Problem Solving from Nature - PPSN V*, pages 292–301. Springer.

[135] Zitzler, E. and Thiele, L. (1999). Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271.

[136] Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., and Grunert da Fonseca, V. (2003). Performance Assessment of Multiobjective Optimizers: An Analysis and Review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132.

Appendix A

Source Code

In this Appendix, we included the source code of the implementation for the algorithms developed, and presented, in this work.

The code is divided in eight files, and each file will be presented as it has been used to perform the experiments performed in this thesis. The files are:

- main.cpp
- mainlib.h
- psolib.h
- fitness-sh2.h
- sigma.h
- fun-moo.h
- variables.h
- randomlib.h

Listing A.1: main.cpp

```
#define PI 3.14159265358979323846264338327950288419716939937 5
unsigned int CONT_FUN = 0;

#include <iostream>
using namespace std;
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "randomlib.h"
#include "fun-moo.h"
#include "variables.h"
#include "fitness-sh2.h"
#include "psolib.h"
#include "sigma.h"
#include "mainlib.h"

int main(int argc, char * const argv[])
{
        unsigned int funcion, particulas, ciclos, optimizacion, MEM;
        char arch1[20];
        char arch2[20];
        clock_t now, later;
        double passed=0.0;
        double ssh;

        if((argc != 6) && (argc != 7))
        {
                cout << "Error, use it as:\n"<<argv[0]<<" funcion particulas
                    ciclos optimizacion[0/1] MEM [ssh]\n";
                exit(0);
        }

        // Function to optimise
        funcion = (unsigned int)atoi(argv[1]);

        // Number of particles
        particulas = (unsigned int)atoi(argv[2]);

        // Number of cycles
        ciclos = (unsigned int)atoi(argv[3]);

        // To minimize = 0, to maximize = 1
        optimizacion = (unsigned int)atoi(argv[4]);

        //***Memory size has to be bigger or equal to the number of
            particles***//
        MEM = (unsigned int)atoi(argv[5]);

        //Sigma sharing
        ssh = 0;

        if(argc == 7)   // If the value for ssh is being manually set
         ssh = (double)atof(argv[6]);

        sprintf(arch1,"plot.txt");
        sprintf(arch2,"ssh.txt");
        now = clock();
        vuelo(funcion, particulas, ciclos, optimizacion, num_dim(funcion),
```

```
		num_fun(funcion), MEM, arch1, arch2, ssh);
	later = clock();
	passed = ( later - now ) / (double)CLOCKS_PER_SEC;
	printf("Total_time_%lf\n", passed);

	return EXIT_SUCCESS;
}
```

Listing A.2: mainlib.h

```
void vuelo(unsigned int fun, unsigned int M, unsigned int Gmax, unsigned int
	opt, unsigned int D, unsigned int NF, unsigned int MEM, char *cad, char
	*ssh_cad, double ssh_line) {

	// Explanation of some variables:

	// fun	-> Function to optimise
	// M		-> Number of particles
	// Gmax -> Number of cycles to run
	// opt	-> type of optimisation (min=0 or max=1)
	// D		-> Number of variables involved in the problem
	// NF	-> Number of functions involved in the problem
	// MEM	-> Size of reposository

	double *pop, *pbests, *vel, *fitness, *fbests;
	double *noDomP, *noDomF;

	pop = new double [M * D];
	pbests = new double [M * D];
	vel = new double [M * D];
	fitness = new double [M * NF];
	fbests = new double [M * NF];
	noDomP = new double [MEM * D];
	noDomF = new double[MEM * NF];

	// pop		-> pointer to the population (variable space)
	// pbests	-> pointer to the location of the best solutions
		found by each particle (variable space)
	// vel		-> pointer to the velocities corresponding to each
		particle
	// fitness	-> pointer to the solutions (objective space)
	// fbests	-> pointer to the best solutions found by each
		particle (objective space)
	// noDomP	-> pointer to the locations of the non dominated
		solutions (variable space)
	// noDomF	-> pointer to the solutions non dominated (objective
		 space)

	unsigned int h, i, j, t, lastPart = 0;

	// h		-> auxiliary variable.
	// i, j		-> counters.
	// t		-> generational counter.
	// lastPart	-> stores the number of particles in the repository.

	double passed;
	clock_t now, later;

	// passed	-> Measures time that it takes to perform all the
		main loop
```

```

double weight, c1, c2, ssh, succ_rate;

double *fShar, *nCounts;
unsigned int *leadC, *leaders;

fShar = new double[MEM];
nCounts = new double[MEM];
leadC = new unsigned int[MEM];
leaders = new unsigned int[M];

// "fShar" and "leadC" had to be the size of the number of particles in the repository,
// because they are the ones being evaluated with fitness sharing.
// On the other hand, "leaders" is the numer of particles in the swarm,
// because that is the leader each particle will follow.

double *distance;
distance = new double[MEM * MEM];

// "distance" is an array which will contain the distance from a particle to the
// rest of the particles in the repository.

FILE *ssh_ot;
ssh_ot = fopen("ssh-ot.txt","w");

////////////////////////////////
//Inicialization of Variables//
////////////////////////////////
for(i = 0; i < (MEM * MEM); i++)
        distance[i] = 0.0;
/*
        Handle the seed range errors
        i = First random number seed must be between 0 and 31328
        j = Second seed must have a value between 0 and 30081
*/
srand((unsigned int)time((time_t *)NULL));
i = (unsigned int) (31329.0 * rand() / (RAND_MAX + 1.0));
j = (unsigned int) (30082.0 * rand() / (RAND_MAX + 1.0));

RandomInitialise(i,j);

now = clock();

// Generation counter, we start at generation 0
t = 0;

// Particles are generated randomly
initpop(pop, fun, M, D);

// Velocities variables are initialized
for(i = 0; i < M; i++)
        for(j = 0; j < D; j++)
                vel[(D * i) + j] = 0.0;

// Particles are evaluated
evaluation(pop, fitness, M, D, fun, NF);

// Their values are copied to memory
copy_array(fitness, fbests, M * NF);
```

```
        copy_array(pop, pbests, M * D);

        // All particles non-dominated are inserted into the repository.
        search_insert(noDomP, noDomF, pop, fitness, D, NF, M, MEM, opt, &lastPart);

        // sigma share is manually set or calculated
        if(ssh_line > 0)
                ssh = ssh_line;
        else
                ssh = get_ssh(noDomF, NF, lastPart);

        // print ssh over time
        fprintf(ssh_ot, "%f_\n", ssh);

        // Calculates fitness sharing for each particle inside the repository
        // generates distances and nCounts
        fitShar(noDomF, lastPart, NF, fShar, MEM, ssh, distance, nCounts);

        // Values used for weight = 0.4 and c1, c2 = 1 (on PSO equation)
        weight = 0.4;
        c1 = 1.0;
        c2 = 1.0;
        succ_rate = 0;

        // Start of the flight
        do {
                //////////////////////////////
                //Calculate 'fitness sharing'//
                //////////////////////////////

                //Assigns the number of times a leader from the repository will be followed
                //(leaders with better fitness sharing will have a greater number of chance to be selected,
                //according to stochastic universal sampling)
                leadAsg(lastPart, fShar, leadC, M, MEM);

                //Each of the particles are assigned a leader from the repository
                //'leaders' is the array holding each of the positions
                leadSel(leadC, leaders, M, MEM);

                for(i = 0; i < M; i++) {
                        // gbest
                        h = leaders[i];

                        // lbest
                        //h = search_best_neighbor(i, fShar, leaders, M);

                        // Selects randomly a leader
                        //h = leadAsg_Random(lastPart);

                        for(j = 0; j < D; j++)
                                vel[(D * i) + j] = velocity(weight, vel[(D * i) + j], c1, c2, pbests[(D * i) + j], noDomP[(D * h) + j], pop[(D * i) + j]);
                }


                //Se calculan las nuevas posiciones
```

```
            for(i = 0; i < M; i++)
                for(j = 0; j < D; j++)
                    pop[(D * i) + j] =      pop[(D * i) + j] +
                        vel[(D * i) + j];

            //Se mantinen las particulas dentro del espacio
            keepin(pop, vel, fun, M, D);

            //Se evalua cada una de las particulas
            evaluation(pop, fitness, M, D, fun, NF);

            search_insert_FS(noDomP, noDomF, pop, fitness, D, NF, M, MEM
                , opt, &lastPart, fShar, ssh, distance, nCounts);

            // fitness sharing
            if(ssh_line > 0)
                ssh = ssh_line;
            else
                ssh = get_ssh(noDomF, NF, lastPart);


            // print ssh over time
            fprintf(ssh_ot, "%f \n", ssh);

            succ_rate = ifbest_interchange_moo(fitness, pop, fbests,
                pbests, M, D, NF, opt);

            // Update the generation counter
            t++;
        } while (Gmax - 1 > t);
        // End of flight

        // Print results
        mo_out(noDomF, lastPart, NF, cad);
        ssh_out(ssh, ssh_cad);

        later = clock();
        passed = ( later - now ) / (double)CLOCKS_PER_SEC;
        printf("Time = %lf\n", passed);
        cout << "End. Bye!" << endl;
        cout << "Sigma Share = " << ssh << endl;
        cout << "c1 = " << c1 << endl;
        cout << "c2 = " << c2 << endl;
        cout << "w = " << weight << endl;
        cout << "SR = " << succ_rate << endl;

        delete [] pop;
        delete [] pbests;
        delete [] vel;
        delete [] fitness;
        delete [] fbests;
        delete [] noDomP;
        delete [] noDomF;
        delete [] leadC;
        delete [] leaders;
        delete [] fShar;
        delete [] nCounts;
        delete [] distance;

        fclose(ssh_ot);
}
```

Listing A.3: psolib.h

```
//Copia el arreglo s a d (De tipo double y longitud l)
void copy_array(double *s, double *d, unsigned int l) {
        unsigned int i;

        for(i = 0; i < l; i++)
                d[i] = s[i];

}

//Busca el valor maximo de un arreglo de tipo double
unsigned int search_max(double *f, unsigned int M) {
        unsigned int i, fmax = 0;

        for(i = 1; i < M; i++)
                if (f[fmax] < f[i])
                        fmax = i;

        return fmax;
}

//Busca el valor maximo de un arreglo de tipo entero sin signo
unsigned int search_max(unsigned int *f, unsigned int M) {
        unsigned int i, fmax = 0;

        for(i = 1; i < M; i++)
                if (f[fmax] < f[i])
                        fmax = i;

        return fmax;
}

//Busca el valor minimo de un arreglo de tipo double
unsigned int search_min(double *f, unsigned int M) {
        unsigned int i, fmin = 0;

        for(i = 1; i < M; i++)
                if (f[fmin] > f[i])
                        fmin = i;

        return fmin;
}

//Busca el valor minimo de un arreglo de tipo entero sin signo
unsigned int search_min(unsigned int *f, unsigned int M) {
        unsigned int i, fmin = 0;

        for(i = 1; i < M; i++)
                if (f[fmin] > f[i])
                        fmin = i;

        return fmin;
}

//Se evaluan las particulas
void evaluation(double *pop, double *fitness, unsigned int M, unsigned int D
    , unsigned int fun, unsigned int NF) {
unsigned int i, j, f;

                for(i = 0; i < M; i++)
                        for(j = 0, f = fun; j < NF; j++, f++)
```

```
                                fitness[(NF * i) + j] = chfun(&pop[D * i], f
                                    );
}

//Calcula la velocidad de las particulas
double velocity(double W, double Vi, double C1, double C2, double Pb, double
     Pg, double Xi) {
        return W * Vi + C1 * RandomDouble(0.0, 1.0) * (Pb - Xi) + C2 *
            RandomDouble(0.0, 1.0) * (Pg - Xi);
}

////////////////////////////////////////////////////////
//MULTI-OBJETIVE
////////////////////////////////////////////////////////

unsigned int best_moo(double *fitness, double *fbests, unsigned int NF,
    unsigned int opt) {
        int best, deflt = 0;
        unsigned int i;
        // Adapted from PAES code:
        for(i = 0; i < NF; i++) {
                if (((fitness[i] < fbests[i]) && (opt == 0)) || ((fitness[i]
                    > fbests[i]) && (opt == 1)))
                        best = 1;
            else
                if (((fitness[i] > fbests[i]) && (opt == 0)) || ((fitness[i]
                    < fbests[i]) && (opt == 1)))
                    best = -1;
                else
                    best = 0;
            if((best) && (best == -deflt))
                return RandomInt(0, 1);
            if(best != 0)
                deflt = best;
    }

    switch (deflt) {
        case 1: i = 1;
                break;
        case -1: i = 0;
                break;
        case 0: i = RandomInt(0, 1);
                break;
    }

        return i;
}

//Returns the number of particles that swaped their old positions for the
    new ones + swaps the positions in their memories
unsigned int ifbest_interchange_moo(double *fitness, double *pop, double *
    fbests, double *pbests, unsigned int M, unsigned int D, unsigned int NF,
     unsigned int opt) {
        unsigned int i, j, cont = 0;

        for(i = 0; i < M; i++) {
                if (best_moo(&fitness[NF * i], &fbests[NF * i], NF, opt)) {
                        for(j = 0; j < NF; j++)
                                fbests[(NF * i) + j] = fitness[(NF * i) + j
                                    ];
                        for(j = 0; j < D; j++)
                                pbests[(D * i) + j] = pop[(D * i) + j];
```

```
                        cont++;
                }
        }
        return cont;
}

// compares two vectors of objective values
// returns:
// 1 if first dominates second,
// 2 if second dominates first,
// 3 otherwise
unsigned int compVec(double *noDomF, unsigned int NF, unsigned int h, unsigned int i, unsigned int opt) {
int best, deflt = 0;
unsigned int j;

        // Adapted from PAES code:
        for(j = 0; j < NF; j++) {
                if (((noDomF[(NF * h) + j] < noDomF[(NF * i) + j]) && (opt == 0)) || ((noDomF[(NF * h) + j] > noDomF[(NF * i) + j]) && (opt == 1)))
                        best = 1;
            else
                if (((noDomF[(NF * h) + j] > noDomF[(NF * i) + j]) && (opt == 0)) || ((noDomF[(NF * h) + j] < noDomF[(NF * i) + j]) && (opt == 1)))
                    best = -1;
                else
                    best = 0;
            if((best) && (best == -deflt))
                return 3;
            if(best != 0)
                deflt = best;
    }

    switch (deflt) {
        case 1: j = 1;
                break;
        case -1: j = 2;
                break;
        case 0: j = 3;
                break;
    }

        return j;
}

void delPartDom(double *noDomP, double *noDomF, unsigned int *lastPart, unsigned int D, unsigned int NF, unsigned int opt) {
        unsigned int h, i, j;
                h = 0;
                i = h + 1;
                do {
                do {
                                if (*lastPart > 1) {
                        switch (compVec(noDomF, NF, h, i, opt)) {
                        //In case 1 and 2 the particles are eliminated (replaced with the last one), because they have been dominated by the h (case 1) or i (case 2)
                                                case 1: for(j = 0; j < D; j++)
                                                        noDomP[(D * i) + j]
```

```
                                                = noDomP[(D * (*lastPart - 1)) + j];
                                        for(j = 0; j < NF; j++)
                                                noDomF[(NF * i) + j] = noDomF[(NF * (*lastPart - 1)) + j];
                                                *lastPart -= 1;
                                break;
                                        case 2: for(j = 0; j < D; j++)
                                                noDomP[(D * h) + j] = noDomP[(D * (*lastPart - 1)) + j];
                                        for(j = 0; j < NF; j++)
                                                noDomF[(NF * h) + j] = noDomF[(NF * (*lastPart - 1)) + j];
                                                *lastPart -= 1;
                                        i = h + 1;
                                break;
                                        case 3: i += 1;
                                break;
                                }
                }
                } while ((i < *lastPart) && (*lastPart > 1));
                h += 1;
                i = h + 1;
                } while ((h < *lastPart - 1) && (*lastPart > 1));
}

// Prints the output
void mo_out(double *noDomF, unsigned int lastPart, unsigned int NF, char *cad) {
        unsigned int i, j;
        FILE *salF;

        salF = fopen(cad,"w");

        for(i = 0; i < lastPart; i++) {
                for(j = 0; j < NF; j++)
                        fprintf(salF, "%f ", noDomF[(NF * i) + j]);
        fprintf(salF, "\n");
        }

        fclose(salF);
}

void search_insert(double *noDomP, double *noDomF, double *pop, double *fitness, unsigned int D, unsigned int NF, unsigned int M, unsigned int MEM, unsigned int opt, unsigned int *lastPart) {
unsigned int i, j, k;

        // Copy all the particles into the repository
        for(i = *lastPart, k = 0; k < M; i++, k++) {
                        for(j = 0; j < D; j++)
                                noDomP[(D * i) + j] = pop[(D * k) + j];
```

```
                        for(j = 0; j < NF; j++) {
                                noDomF[(NF * i) + j] = fitness[(NF * k) + j
                                    ];
                        }
                        *lastPart += 1;
        }

        // deletes dominated particles
        delPartDom(noDomP, noDomF, lastPart, D, NF, opt);

}

// compares two vectors of objective values
// returns:
// 1 if first dominates second,
// 2 if second dominates first,
// 3 otherwise
unsigned int compVec2(double *noDomF, double *fitness, unsigned int NF,
    unsigned int h, unsigned int i, unsigned int opt) {
int best, deflt = 0;
unsigned int j;

        // Adapted from PAES code:
        for(j = 0; j < NF; j++) {
                if (((noDomF[(NF * h) + j] < fitness[(NF * i) + j]) && (opt
                    == 0)) || ((noDomF[(NF * h) + j] > fitness[(NF * i) + j
                    ]) && (opt == 1)))
                        best = 1;
           else
                if (((noDomF[(NF * h) + j] > fitness[(NF * i) + j]) && (opt
                    == 0)) || ((noDomF[(NF * h) + j] < fitness[(NF * i) + j])
                    && (opt == 1)))
                    best = -1;
                else
                    best = 0;
           if((best) && (best == -deflt))
                return 3;
           if(best != 0)
                deflt = best;
    }

    switch (deflt) {
        case 1: j = 1;
                break;
        case -1: j = 2;
                break;
        case 0: j = 3;
                break;
        }

        return j;
}

//Search for the boundaries of every dimension in the objective space
void search_limits(double *noDomF, unsigned int *lsup, unsigned int *linf,
    unsigned int MEM, unsigned int NF, unsigned int lastPart)
{
  double *aux;
  unsigned int i, j;
  aux = new double [MEM];

  for(j = 0; j < NF; j++)
```

```
  {
    for(i = 0; i < lastPart; i++)
      aux[i] = noDomF[(NF * i) + j];
    linf[j] = (NF * search_min(aux, i)) + j;
    lsup[j] = (NF * search_max(aux, i)) + j;
  }
  delete [] aux;
}

// Euclidean distance between particle 'i' and particle 'j'
// Note: In N dimensions, the Euclidean distance between two points p and q
    is:
// sqrt(sum(p_i - q_i)^2) where p_i (or q_i) is the coordinate of p (or q)
    in dimension i.
double dCount(unsigned int i, unsigned int j, double *noDomF, unsigned int
    NF)
{
  unsigned int d;
  double sum = 0.0;

  for(d = 0; d < NF; d++)
        sum += pow((noDomF[(NF * i) + d] - noDomF[(NF * j) + d]), 2);

  return sqrt(sum);
}

// Assigns a fitness sharing to a particle according to the distance between
     it and the others.
double sharing(unsigned int i, unsigned int j, double *noDomF, unsigned int
    NF, double ssh, double *distance, unsigned int MEM)
{
  double d, r = 0.0;
  int alp = 2;

  d = dCount(i, j, noDomF, NF);
  distance[(MEM * i) + j] = d;
  if (d < ssh)
    r = 1.0 - pow(d / ssh, alp);
  return r;
}

// Adds fitness sharing to a single particle 'i' in accordance to the
    closeness to the rest
// Basicaly calculates the distance for each particle 'j' and if too close
    adds fitness sharing to the particle 'i'
double nCount(unsigned int i, double *noDomF, unsigned int lastPart,
    unsigned int NF, double ssh, double *distance, unsigned int MEM)
{
  double sum = 0.0;
  unsigned int j;

  for(j = 0; j < lastPart; j++)
    sum += sharing(i, j, noDomF, NF, ssh, distance, MEM);

  return sum;
}

// Assings a fitness sharing to every particle in the repository
// The fitness sharing values for particles lying in 'crowded' areas will be
     lower,
// than particles lying in 'lonely' areas where fitness sharing values will
    be higher
```

```
void fitShar(double *noDomF, unsigned int lastPart, unsigned int NF, double
    *fShar, unsigned int MEM, double ssh, double *distance, double *nCounts)
{
  unsigned int i;
  double ffix = 10.0;

  for(i = 0; i < lastPart; i++) {
        nCounts[i] = nCount(i, noDomF, lastPart, NF, ssh, distance, MEM);
        fShar[i] = ffix / nCounts[i];
  }
}

// Selection of a leader acording to the best FS
unsigned int leadAsg_BestOfRep(double *fShar, unsigned int lastPart)
{
        return search_max(fShar, lastPart);
}

// Selection of a random leader
unsigned int leadAsg_Random(unsigned int lastPart)
{
        return RandomInt(0, lastPart);
}

// Choose the particle leaders, according to a SUS (stochastic universal
    sampling)
// The number of times each leader will be used is stored in the array leadC
    ,
// which corresponds to the positions of the particles in the repository
void leadAsg(unsigned int lastPart, double *fShar, unsigned int *leadC,
    unsigned int M, unsigned int MEM) {
  unsigned int i;
  double fSharS = 0.0, *Pr, U, sum = 0.0;
  Pr = new double[lastPart];

  for(i = 0; i < lastPart; i++)
    fSharS += fShar[i];

  for(i = 0; i < lastPart; i++)
    Pr[i] = fShar[i] / fSharS;

  U = RandomDouble(0.0, (double) 1/M);

  for(i = 0; i < lastPart; i++)
  {
    leadC[i] = 0;
    sum += Pr[i];

    while (U < sum)
    {
      leadC[i] += 1;
      U += (double) 1/M;
    }
  }

  for(i = i; i < MEM; i++)
    leadC[i] = 0;

  delete [] Pr;

}
```

```
//Reduces the number of times a particle can be choosen as a leader
void leadSel(unsigned int *leadC, unsigned int *leaders, unsigned int M,
    unsigned int MEM)
{
  unsigned int i = 0, j = 0;

        do {
                while ((leadC[j] > 0) && (i < M))
                {
                        leaders[i] = j;
                        leadC[j] -= 1;
                        i++;
                }
                j++;
        } while ((i < M) && (j < MEM));
}

void update_dist(double *distance, unsigned int MEM, unsigned int lastPart,
    unsigned int h) {

        unsigned int i = 0;
        // Update the distances swaping the last in the array with the h
            distance position
        // when the position h is found then the distances have to be shift
        while (i < h)
        {
                distance[(MEM * h) + i] = distance[(MEM * (lastPart - 1)) +
                    i];
                i++;
        }
        // A distance of 0 is set for the position h
        distance[(MEM * h) + i] = 0;
        // Shifts, and the swapping carries on
        for(i = i + 1;  i < lastPart; i++)
                distance[(MEM * h) + i] = distance[(MEM * (lastPart - 1)) +
                    i];

        // Once the swapping is finished, is necesary to change the distance
             values for the particle
        // swapped in the other arrays.
        // Also the values for the distance which is being swapped is
            deleted (to 0)
        for(i = 0; i < lastPart; i++) {
                distance[(MEM * i) + h] = distance[(MEM * h) + i];
                // The distances just swapped are deleted
                distance[(MEM * i) + (lastPart - 1)] = 0;
                distance[(MEM * (lastPart - 1)) + i] = 0;
        }

}

void update_fs(double *distance, double *fShar, double *nCounts, unsigned
    int MEM, unsigned int lastPart, unsigned int h, double ssh) {
        int alp = 2;
        unsigned int i;
        double ffix = 10.0;

        for(i = 0; i < lastPart; i++)
                if((distance[(MEM * i) + h] < ssh) && (i != h)) {
                        // Updates if distance of particle h to be deleted
                            affects particle i, when their distance is
                            smaller than ssh
```

```
                nCounts[i] = nCounts[i] - (1.0 - pow(distance[(MEM * i) + h] / ssh, alp));
                fShar[i] = ffix / nCounts[i];
            }

}

unsigned int verDom2(double *noDomP, double *noDomF, double *fitness, unsigned int D, unsigned int NF, unsigned int opt,
                                        unsigned int *lastPart, unsigned int i, unsigned int MEM, double *distance, double *fShar,
                                        double *nCounts, double ssh) {
unsigned int h = 0, j;
do {
        switch (compVec2(noDomF, fitness, NF, h, i, opt)) {
                // If noDomF dominates fitness, case 1.
                case 1: return 0;
                // If noDomF is dominated by fitness, case 2.
                case 2:
                        // Particle noDom dominated is deleted from the repository
                        for(j = 0; j < D; j++)
                                noDomP[(D * h) + j] = noDomP[(D * (*lastPart - 1)) + j];
                        for(j = 0; j < NF; j++)
                                noDomF[(NF * h) + j] = noDomF[(NF * (*lastPart - 1)) + j];
                        // Distances and fitness sharing for each updated for each particle
                        update_fs(distance, fShar, nCounts, MEM, *lastPart, h, ssh);
                        update_dist(distance, MEM, *lastPart, h);
                        nCounts[h] = nCounts[*lastPart - 1];
                        fShar[h] = fShar[*lastPart - 1];
                        // Number of particles in the repository is decresead by one
                        *lastPart -= 1;
                        break;
                case 3: h += 1;
                        break;
        }
} while(h < *lastPart);

        return 1;
}

void distance_fShar_update(unsigned int MEM, double *distance, double *fShar, double *nCounts, double ssh, unsigned int lastPart) {
        int alp = 2;
        unsigned int i, last = lastPart - 1;
        double ffix = 10.0;

        // Update distances between the new particle and the rest
        for(i = 0; i < last; i++)
                distance[(MEM * i) + last] = distance[(MEM * last) + i];

        // Update fitShar if necessary
        for(i = 0; i < last; i++)
                if(distance[(MEM * i) + last] < ssh) {
                        // Update
                        nCounts[i] = nCounts[i] + (1.0 - pow(distance[(MEM *
```

```c
                                i) + last] / ssh, alp));
                        fShar[i] = ffix / nCounts[i];
                }
}

void search_insert_FS(double *noDomP, double *noDomF, double *pop, double *fitness,
                                        unsigned int D, unsigned int NF, unsigned int M, unsigned int MEM,
                                        unsigned int opt, unsigned int *lastPart, double *fShar, double ssh,
                                        double *distance, double *nCounts) {

        unsigned int i, j, k, l;
        double aux_1, aux_2, ffix = 10.0;

        for(k = 0; k < M; k++) {
                // Verify if the particle aiming for the repositoy is NOT dominated by the ones inside,
                // and it does deletes the ones it manages to dominate.
                if (verDom2(noDomP, noDomF, fitness, D, NF, opt, lastPart, k, MEM, distance, fShar, nCounts, ssh) == 1) {
                        // If the repository is not full, it does insert the particle, updating fitness sharing
                if (*lastPart < MEM) {
                        i = *lastPart;
                        for(j = 0; j < D; j++)
                                noDomP[(D * i) + j] = pop[(D * k) + j];
                        for(j = 0; j < NF; j++)
                                noDomF[(NF * i) + j] = fitness[(NF * k) + j];

                                // Update the counter of particles in the repository
                                *lastPart += 1;

                                // Update fitness sharing for each particle
                                // First we calculate distances (along with fShar and nCounts) to see if it's closer to the particles
                                nCounts[i] = nCount(i, noDomF, *lastPart, NF, ssh, distance, MEM);
                                fShar[i] = ffix / nCounts[i];
                                // With the distances we can now see if we need to recalculate the distances and fShar of each particle or not
                                distance_fShar_update(MEM, distance, fShar, nCounts, ssh, *lastPart);
                        }
                        // When the particle is not dominated, and the repository is full
                        // to insert it into the repository, the particle needs a better FS than one of them
                else {
                                // calculate fitness sharing of particles
                                aux_1 = calFS(noDomF, fitness, *lastPart, NF, MEM, ssh, k);
                                l = search_min(fShar, MEM);
                                aux_2 = fShar[l];
```

```
                        // If particle k->(aux_1) has better FS than particle l->(aux_2), the latter is deleted
                    if (aux_1 > aux_2) {
                            // We eliminate the particle with the smaller FS
                            for(j = 0; j < D; j++)
                                noDomP[(D * l) + j] = noDomP[(D * (*lastPart - 1)) + j];
                            for(j = 0; j < NF; j++)
                                noDomF[(NF * l) + j] = noDomF[(NF * (*lastPart - 1)) + j];
                            // We update distances for each particle, along their fitness sharing values
                            update_fs(distance, fShar, nCounts, MEM, *lastPart, l, ssh);
                            update_dist(distance, MEM, *lastPart, l);
                            nCounts[l] = nCounts[*lastPart - 1];
                            fShar[l] = fShar[*lastPart - 1];
                            // The number of particles in the repository is decremented by one
                            *lastPart -= 1;

                            // The particle with better FS is inserted
                            i = *lastPart;
                            for(j = 0; j < D; j++)
                                noDomP[(D * i) + j] = pop[(D * k) + j];
                            for(j = 0; j < NF; j++)
                                noDomF[(NF * i) + j] = fitness[(NF * k) + j];

                            // Update the counter of particles in the repository
                            *lastPart += 1;

                            // Update FS for each particle
                            // First we calculate distances (along with fShar and nCounts) to see if it's closer to the particles
                            nCounts[i] = nCount(i, noDomF, *lastPart, NF, ssh, distance, MEM);
                            fShar[i] = ffix / nCounts[i];
                            // With the distances we can now see if we need to recalculate the fShar of each particle or not
                            distance_fShar_update(MEM, distance, fShar, nCounts, ssh, *lastPart);
                        }
                }
        }
        }
}
```

```
unsigned int search_best_neighbor(unsigned int i, double *fitness, unsigned
    int *leaders, unsigned int M) {

        unsigned int x, y;

        if (i > 0)
                x = i - 1;
        else
                x = M - 1;

        if (i >= M - 1)
                y = 0;
        else
                y = i + 1;

        if (fitness[leaders[x]] > fitness[leaders[y]]) {
                if (fitness[leaders[x]] > fitness[leaders[i]])
                        return x;
                else
                        return i;
        }
        else {
                if (fitness[leaders[i]] > fitness[leaders[y]])
                        return i;
                else
                        return y;
        }
}

void ssh_out(double ssh, char *cad) {

        FILE *sal;

        sal = fopen(cad,"w");

        fprintf(sal, "%f \n", ssh);

        fclose(sal);
}

// Inertia weight(see ratnaweera et al., IEEE trans. EC, vol. 8, no. 3)
double inertia_weight(double w1, double w2, unsigned int MAXITER, unsigned
    int iter) {

return (w1 - w2) * ((double)(MAXITER - iter) /  (double)MAXITER) + w2;

}

double acc_coef(double ci, double cf, unsigned int MAXITER, unsigned int
    iter) {

return ((cf - ci) * (iter /  MAXITER) + ci);

}

double acc_coef1(double cont, unsigned int M) {
        double rate;

        rate = cont / M;

        // If the performance rate is high we need to reinforce the
            cognitive component
```

```
        if(rate >= 0.6)
                return 1/0.9;//0.01;
        else
        // On the other hand if the performance rate is low we need to
            decrease the cognitive component
                if(rate <= 0.4)
                        return 0.9;//-0.01;

        return 1.0;

}

double acc_coef2(double cont, unsigned int M) {
        double rate;

        rate = cont / M;

        // If the performance rate is high we need to decrease the social
            component
        if(rate >= 0.6)
                return 0.9;//-0.01;
        else
        // On the other hand if the performance rate is low we need to
            reinforce the social component
                if(rate <= 0.4)
                        return 1/0.9;//0.01;

        return 1.0;

}
```

Listing A.4: fitness-sh2.h

```
        for(d = 0; d < NF; d++)
                sum += pow((fitness[(NF * i) + d] - noDomF[(NF * j) + d]),
                    2);

        return sqrt(sum);
}

//Assigns a fitness sharing to a particle according to the distance between
    it and the others.
double sharing2(unsigned int i, unsigned int j, double *noDomF, double *
    fitness, unsigned int NF, double ssh)
{
        double d, r = 0.0;//, ssh = 4.5;
        int alp = 2;

        d = dCount2(i, j, noDomF, fitness, NF);
        //cout << d << endl;
        if (d < ssh)
                r = 1.0 - pow(d / ssh, alp);
        //cout << r << endl;
        return r;
}

//Adds fitness sharing to a single particle 'l' in accordance to the
    closeness to the rest
//Basicaly calculates the distance for each particle 'i' and if too close
    adds fitness sharing to the particle 'l'
```

```
double nCount2(unsigned int i, double *noDomF, double *fitness, unsigned int
      lastPart, unsigned int NF, double ssh)
{
        double sum = 0.0;
        unsigned int j;

        for(j = 0; j < lastPart; j++)
                sum += sharing2(i, j, noDomF, fitness, NF, ssh);

        //+ 1.0 to add the sharing not counted for that particle.
        return sum + 1.0;
}

//Assings a fitness sharing to a given particle according to the particles
     in the repository
//The fitness sharing value for a particle lying in a 'crowded' area will be
      lower,
//than a particles lying in 'lonely' areas where fitness sharing values will
      be higher
double calFS(double *noDomF, double *fitness, unsigned int lastPart,
     unsigned int NF, unsigned int MEM, double ssh, unsigned int i)
{

        double res, ffix = 10.0;
        //search_limits(noDomF, lsup, linf, MEM, NF, lastPart);
        res = ffix / nCount2(i, noDomF, fitness, lastPart, NF, ssh);
        //for(l = 0; l < lastPart; l++)
        return res;
}
```

Listing A.5: sigma.h

```

    //ifstream archivo1(argv[1], ios::in);

    for(i = 0; i < NF; i++) {
        max[i] = -999999999;
        min[i] = 999999999;
    }

    for(i = 0; i < lastPart; i++) {
                for(j = 0; j < NF; j++) {
                        value = noDomF[(NF * i) + j];
            max[j] = max[j] < value ? value : max[j];
            min[j] = min[j] > value ? value : min[j];
        }
    }


    for(i = 0; i < NF; i++)
        sum += pow(max[i] - min[i], 2);

    MD = sqrt(sum);

        delete [] max;
        delete [] min;

        return MD;
}

```

```
double get_ssh(double *noDomF, unsigned int NF, unsigned int lastPart) {
        double MD, ssh;

        MD = get_MD(noDomF, NF, lastPart);

        ssh = MD / lastPart;

        return ssh;
}
```

Listing A.6: fun-moo.h

```

//Deb DTLZ1

double dtlz_1_1(double *x) {
        unsigned int i, k = 5, n = 7;
        double g = 0.0;

    for(i = n - k; i < n; i++)
                g += pow(x[i] - 0.5, 2) - cos(20.0 * PI * (x[i] - 0.5));

        g = 100.0 * (k + g);

        return 0.5 * x[0] * x[1] * (1.0 + g);
}

double dtlz_1_2(double *x) {
        unsigned int i, k = 5, n = 7;
        double g = 0.0;

    for(i = n - k; i < n; i++)
                g += pow(x[i] - 0.5, 2) - cos(20.0 * PI * (x[i] - 0.5));

        g = 100.0 * (k + g);

        return 0.5 * x[0] * (1.0 - x[1]) * (1.0 + g);
}

double dtlz_1_3(double *x) {
        unsigned int i, k = 5, n = 7;
        double g = 0.0;

    for(i = n - k; i < n; i++)
                g += pow(x[i] - 0.5, 2) - cos(20.0 * PI * (x[i] - 0.5));

        g = 100.0 * (k + g);

        return 0.5 * (1.0 - x[0]) * (1.0 + g);
}

//Deb DTLZ2

double dtlz_2_1(double *x) {
        unsigned int i, n = 12, k = 10;
        double sum = 0.0;

    for(i = n - k; i < n; i++)
                sum += pow(x[i] - 0.5, 2);

```

```
        return (1.0 + sum) * cos(x[0] * PI / 2.0) * cos(x[1] * PI / 2.0);
}

double dtlz_2_2(double *x) {
        unsigned int i, n = 12, k = 10;
        double sum = 0.0;

    for(i = n - k; i < n; i++)
                sum += pow(x[i] - 0.5, 2);

        return (1.0 + sum) * cos(x[0] * PI / 2.0) * sin(x[1] * PI / 2.0);
}

double dtlz_2_3(double *x) {
        unsigned int i, n = 12, k = 10;
        double sum = 0.0;

    for(i = n - k; i < n; i++)
                sum += pow(x[i] - 0.5, 2);

        return (1.0 + sum) * sin(x[0] * PI / 2.0);
}

//Deb DTLZ3

double dtlz_3_1(double *x) {
        unsigned int i, n = 12, k = 10;
        double sum = 0.0;

    for(i = n - k; i < n; i++)
                sum += pow(x[i] - 0.5, 2) - cos(20.0 * PI * (x[i] - 0.5));

        sum = 100.0 * (k + sum);

        return (1.0 + sum) * cos(x[0] * PI / 2.0) * cos(x[1] * PI / 2.0);
}

double dtlz_3_2(double *x) {
        unsigned int i, n = 12, k = 10;
        double sum = 0.0;

    for(i = n - k; i < n; i++)
                sum += pow(x[i] - 0.5, 2) - cos(20.0 * PI * (x[i] - 0.5));

        sum = 100.0 * (k + sum);

        return (1.0 + sum) * cos(x[0] * PI / 2.0) * sin(x[1] * PI / 2.0);
}

double dtlz_3_3(double *x) {
        unsigned int i, n = 12, k = 10;
        double sum = 0.0;

    for(i = n - k; i < n; i++)
                sum += pow(x[i] - 0.5, 2) - cos(20.0 * PI * (x[i] - 0.5));

        sum = 100.0 * (k + sum);

        return (1.0 + sum) * sin(x[0] * PI / 2.0);
}

//Deb DTLZ4
```

```
double dtlz_4_1(double *x) {
        double g = 0, alpha = 100;
        unsigned int i, n = 12, k = 10;

        for (i = n - k; i < n; i++)
                g += pow(x[i] - 0.5, 2);

        return (1.0 + g) * cos(pow(x[0], alpha) * PI / 2) * cos(pow(x[1],
            alpha) * PI / 2);

}

double dtlz_4_2(double *x) {
        double g = 0, alpha = 100;
        unsigned int i, n = 12, k = 10;

        for (i = n - k; i < n; i++)
                g += pow(x[i] - 0.5, 2);

        return (1.0 + g) * cos(pow(x[0], alpha) * PI / 2) * sin(pow(x[1],
            alpha) * PI / 2);

}

double dtlz_4_3(double *x) {
        double g = 0, alpha = 100;
        unsigned int i, n = 12, k = 10;

        for (i = n - k; i < n; i++)
                g += pow(x[i] - 0.5, 2);

        return (1.0 + g) * sin(pow(x[0], alpha) * PI / 2);

}

//Deb DTLZ5

double dtlz_5_1(double *x) {
        unsigned int i, n = 12, k = 10, dim = 3;
        double g = 0, t, theta[dim];

        for(i = n - k; i < n; i++)
                g += pow(x[i] - 0.5, 2);

        t = PI / (4 * (1 + g));

        theta[0] = x[0] * PI / 2;

        for(i = 1; i < dim - 1; i++)
                theta[i] = t * (1 + 2 * g * x[i]);

        return (1 + g) * cos(theta[0]) * cos(theta[1]);
}

double dtlz_5_2(double *x) {
        unsigned int i, n = 12, k = 10, dim = 3;
        double g = 0, t, theta[dim];

        for(i = n - k; i < n; i++)
                g += pow(x[i] - 0.5, 2);

```

```
        t = PI / (4 * (1 + g));

        theta[0] = x[0] * PI / 2;

        for(i = 1; i < dim - 1; i++)
                theta[i] = t * (1 + 2 * g * x[i]);

        return (1 + g) * cos(theta[0]) * sin(theta[1]);
}

double dtlz_5_3(double *x) {
        unsigned int i, n = 12, k = 10, dim = 3;
        double g = 0, t, theta[dim];

        for(i = n - k; i < n; i++)
                g += pow(x[i] - 0.5, 2);

        t = PI / (4 * (1 + g));

        theta[0] = x[0] * PI / 2;

        for(i = 1; i < dim - 1; i++)
                theta[i] = t * (1 + 2 * g * x[i]);

        return (1 + g) * sin(theta[0]);
}

//Deb DTLZ6

double dtlz_6_1(double *x) {
        unsigned int i, n = 12, k = 10, dim = 3;
        double g = 0, t, theta[dim];

        for(i = n - k; i < n; i++)
                g += pow(x[i], 0.1);

        t = PI / (4 * (1 + g));

        theta[0] = x[0] * PI / 2;

        for(i = 1; i < dim - 1; i++)
                theta[i] = t * (1 + 2 * g * x[i]);

        return (1 + g) * cos(theta[0]) * cos(theta[1]);
}

double dtlz_6_2(double *x) {
        unsigned int i, n = 12, k = 10, dim = 3;
        double g = 0, t, theta[dim];

        for(i = n - k; i < n; i++)
                g += pow(x[i], 0.1);

        t = PI / (4 * (1 + g));

        theta[0] = x[0] * PI / 2;

        for(i = 1; i < dim - 1; i++)
                theta[i] = t * (1 + 2 * g * x[i]);

        return (1 + g) * cos(theta[0]) * sin(theta[1]);
}
```

```
double dtlz_6_3(double *x) {
        unsigned int i, n = 12, k = 10, dim = 3;
        double g = 0, t, theta[dim];

        for(i = n - k; i < n; i++)
                g += pow(x[i], 0.1);

        t = PI / (4 * (1 + g));

        theta[0] = x[0] * PI / 2;

        for(i = 1; i < dim - 1; i++)
                theta[i] = t * (1 + 2 * g * x[i]);

        return (1 + g) * sin(theta[0]);
}

//Deb DTLZ7

double dtlz_7_1(double *x) {

        return x[0];
}

double dtlz_7_2(double *x) {

        return x[1];
}

double dtlz_7_3(double *x) {
        double g = 0, h = 0;
        unsigned int i, n = 22, k = 20, M = 3;

        for(i = n - k; i < n; i++)
                g += x[i];

        g = 1.0 + 9.0 * g / k;          // From PISA
        //g = 1.0 + (9.0 / k) * g;      // From Deb
        //g = 1.0 + (9.0 / n) * g;      // From Coello

        for(i = 0; i < M - 1; i++)
                h += (x[i] / (1.0 + g)) * (1.0 + sin(3.0 * PI * x[i]));

        h = M - h;

        return (1.0 + g) * h;
}

//Veldhuizen MOP1

double veld_1_1(double *x) {
        return pow(x[0], 2);
}

double veld_1_2(double *x) {
        return pow(x[0] - 2.0, 2);
}

//Veldhuizen MOP2

double veld_2_1(double *x) {
```

```
        double r = 0.0;
        unsigned int i;

        for(i = 0; i < 3; i++)
                r += pow(x[i] - (1.0 / sqrt(3)), 2);

        return (1.0 - exp(-r));
}

double veld_2_2(double *x) {
        double r = 0.0;
        unsigned int i;

        for(i = 0; i < 3; i++)
                r += pow(x[i] + (1.0 / sqrt(3)), 2);

        return (1.0 - exp(-r));
}

//Veldhuizen MOP3

double veld_3_1(double *x) {
        double A1, A2, B1, B2;

        A1 = 0.5 * sin(1.0) - 2.0 * cos(1.0) + sin(2.0) - 1.5 * cos(2.0);
        A2 = 1.5 * sin(1.0) - cos(1.0) + 2.0 * sin(2.0) - 0.5 * cos(2.0);
        B1 = 0.5 * sin(x[0]) - 2.0 * cos(x[0]) + sin(x[1]) - 1.5 * cos(x[1])
            ;
        B2 = 1.5 * sin(x[0]) - cos(x[0]) + 2.0 * sin(x[1]) - 0.5 * cos(x[1])
            ;

        return 1.0 * -(1.0 + pow(A1 - B1, 2) + pow(A2 - B2, 2) );
}

double veld_3_2(double *x) {
        return 1.0 * -(pow(x[0] + 3.0, 2) + pow(x[1] + 1.0, 2) );
}

//Veldhuizen MOP4

double veld_4_1(double *x) {
        double r = 0.0;
        unsigned int i;

        for(i = 0; i < 2; i++)
                r += -10.0 * exp(-0.2 * sqrt(pow(x[i], 2) + pow(x[i + 1], 2)
                    ) );

        return r;
}

double veld_4_2(double *x) {
        double r = 0.0;
        unsigned int i;

        for(i = 0; i < 3; i++)
                r += pow(fabs(x[i]), 0.8) + 5.0 * sin(pow(x[i], 3));
        return r;
}

//Veldhuizen MOP5

```

```
double veld_5_1(double *x) {
        return 0.5 * (pow(x[0], 2) + pow(x[1], 2)) + sin(pow(x[0], 2) + pow(x[1], 2));
}

double veld_5_2(double *x) {
        return (pow(3.0 * x[0] - 2.0 * x[1] + 4.0, 2) / 8.0) + (pow(x[0] - x[1] + 1.0, 2) / 27.0) + 15.0;
}

double veld_5_3(double *x) {
        return (1 / (pow(x[0], 2) + pow(x[1], 2) + 1.0)) - 1.1 * exp(-pow(x[0], 2) - pow(x[1], 2));
}

//Veldhuizen MOP6

double veld_6_1(double *x) {
        return x[0];
}

double veld_6_2(double *x) {
        double a, q;
        a = 2.0;
        q = 4.0;

        return (1.0 + 10.0 * x[1]) * (1.0 - pow((x[0] / (1.0 + 10.0 * x[1])), a) - (x[0] / (1.0 + 10.0 * x[1])) * sin(2.0 * PI * q * x[0]));
}


//Veldhuizen MOP7

double veld_7_1(double *x) {
        return (pow(x[0] - 2.0, 2) / 2.0) + (pow(x[1] + 1.0, 2) / 13.0) + 3.0;
}

double veld_7_2(double *x) {
        return (pow(x[0] + x[1] - 3.0, 2) / 36.0) + (pow(-x[0] + x[1] + 2.0, 2) / 8.0) - 17.0;
}

double veld_7_3(double *x) {
        return (pow(x[0] + 2 * x[1] - 1.0, 2) / 175.0) + (pow(2.0 * x[1] - x[0], 2) / 17.0) - 13.0;
}

//ZDT1

double zdt_1_1(double *x) {
        return x[0];
}

double zdt_1_2(double *x) {
        int i, n = 30;
        double g, h, sum = 0.0;

        for(i = 1; i < n; i++)
                sum += x[i] / ( n - 1.0);
        g = 1.0 + (9.0 * sum);
    h = 1.0 - sqrt(x[0] / g);
```

```

    return g * h;
}

//ZDT2

double zdt_2_1(double *x) {
        return x[0];
}

double zdt_2_2(double *x) {
        int i, n = 30;
        double g, h, sum = 0.0;

        for(i = 1; i < n; i++)
                sum += x[i] / ( n - 1.0);
        g = 1.0 + (9.0 * sum);
    h = 1.0 - pow(x[0] / g, 2);

    return g * h;
}

//ZDT3

double zdt_3_1(double *x) {
        return x[0];
}

double zdt_3_2(double *x) {
        int i, n = 30;
        double g, h, sum = 0.0;

        for(i = 1; i < n; i++)
                sum += x[i] / (n - 1.0);
        g = 1.0 + (9.0 * sum);
    h = 1.0 - sqrt(x[0] / g) - (x[0] / g) * sin(10.0 * PI * x[0]);

    return g * h;
}

//ZDT4

double zdt_4_1(double *x) {
        return x[0];
}

double zdt_4_2(double *x) {
        int i, n = 10;
        double g, h, sum = 0.0;

        for(i = 1; i < n; i++)
                sum += pow(x[i], 2) - (10.0 * cos(4 * PI * x[i]));
        g = 1.0 + (10.0 * (10.0 - 1.0)) + sum;
    h = 1.0 - sqrt(x[0] / g);

    return g * h;
}

//ZDT6

double zdt_6_1(double *x) {

```

```
        double value = 1.0 - exp(-4.0 * x[0]) * pow(sin(6.0 * PI * x[0]),6);

        return value;
}

double zdt_6_2(double *x) {
        int n = 10;

        double f1, g, h, sum = 0.0;

        f1 = 1.0 - exp(-4.0 * x[0]) * pow(sin(6.0 * PI * x[0]),6);
        for(int i = 1; i < n; i++)
                sum += x[i]/9.0;
        g = 1.0 + 9.0 * pow(sum, 0.25);
        h = 1.0 - ((f1/g)*(f1/g));

        return g * h;
}
```

Listing A.7: variables.h

```
unsigned int num_fun(unsigned int fun){

        unsigned int num = 1;

        switch (fun) {
                case 100: num = 2; break;
                case 200: num = 2; break;
                case 300: num = 2; break;
                case 400: num = 2; break;
                case 500: num = 3; break;
                case 600: num = 2; break;
                case 700: num = 3; break;

                case 1100: num = 3; break;
                case 1200: num = 3; break;
                case 1300: num = 3; break;
                case 1400: num = 3; break;
                case 1500: num = 3; break;
                case 1600: num = 3; break;
                case 1700: num = 3; break;

                case 2100: num = 2; break;
                case 2200: num = 2; break;
                case 2300: num = 2; break;
                case 2400: num = 2; break;
                case 2600: num = 2; break;
        }

        return num;

}

unsigned int num_dim(unsigned int fun){

        unsigned int num = 0;

        switch (fun) {

                case 100: num = 1; break;
```

```
                case 200: num = 3; break;
                case 300: num = 2; break;
                case 400: num = 3; break;
                case 500: num = 2; break;
                case 600: num = 2; break;
                case 700: num = 2; break;

                case 1100: num = 7; break;
                case 1200: num = 12; break;
                case 1300: num = 12; break;
                case 1400: num = 12; break;
                case 1500: num = 12; break;
                case 1600: num = 12; break;
                case 1700: num = 22; break;

                case 2100: num = 30; break;
                case 2200: num = 30; break;
                case 2300: num = 30; break;
                case 2400: num = 10; break;
                case 2600: num = 10; break;
        }

        return num;

}

void initpop(double *pop, unsigned int fun, unsigned int M, unsigned int D)
    {

        unsigned int i, j;

        switch (fun) {

                case 100:
                        for(i = 0; i < M; i++)
                                pop[(D * i) + 0] = RandomDouble(-100000.0,
                                    100000.0);
                        break;
                case 200:
                        for(i = 0; i < M; i++)
                                for(j = 0; j < 3; j++)
                                        pop[(D * i) + j] = RandomDouble
                                            (-4.0, 4.0);
                        break;
                case 300:
                        for(i = 0; i < M; i++)
                                for(j = 0; j < 2; j++)
                                        pop[(D * i) + j] = RandomDouble
                                            (-3.1416, 3.1416);
                        break;
                case 400:
                        for(i = 0; i < M; i++)
                                for(j = 0; j < 3; j++)
                                        pop[(D * i) + j] = RandomDouble
                                            (-5.0, 5.0);
                        break;
                case 500:
                        for(i = 0; i < M; i++)
                                for(j = 0; j < 2; j++)
                                        pop[(D * i) + j] = RandomDouble
                                            (-3.0, 3.0);
                        break;
```

```
                case 600:
                    for(i = 0; i < M; i++)
                        for(j = 0; j < 2; j++)
                            pop[(D * i) + j] = RandomDouble(0.0,
                                1.0);
                    break;
                case 700:
                    for(i = 0; i < M; i++)
                        for(j = 0; j < 2; j++)
                            pop[(D * i) + j] = RandomDouble
                                (-4.0, 4.0);
                    break;


                case 1100:
                    for(i = 0; i < M; i++)
                        for(j = 0; j < 7; j++)
                            pop[(D * i) + j] = RandomDouble(0.0,
                                1.0);
                    break;
                case 1200:
                    for(i = 0; i < M; i++)
                        for(j = 0; j < 12; j++)
                            pop[(D * i) + j] = RandomDouble(0.0,
                                1.0);
                    break;
                case 1300:
                    for(i = 0; i < M; i++)
                        for(j = 0; j < 12; j++)
                            pop[(D * i) + j] = RandomDouble(0.0,
                                1.0);
                    break;
                case 1400:
                    for(i = 0; i < M; i++)
                        for(j = 0; j < 12; j++)
                            pop[(D * i) + j] = RandomDouble(0.0,
                                1.0);
                    break;
                case 1500:
                    for(i = 0; i < M; i++)
                        for(j = 0; j < 12; j++)
                            pop[(D * i) + j] = RandomDouble(0.0,
                                1.0);
                    break;
                case 1600:
                    for(i = 0; i < M; i++)
                        for(j = 0; j < 12; j++)
                            pop[(D * i) + j] = RandomDouble(0.0,
                                1.0);
                    break;
                case 1700:
                    for(i = 0; i < M; i++)
                        for(j = 0; j < 22; j++)
                            pop[(D * i) + j] = RandomDouble(0.0,
                                1.0);
                    break;


                case 2100:
                    for(i = 0; i < M; i++)
                        for(j = 0; j < 30; j++)
                            pop[(D * i) + j] = RandomDouble(0.0,
```

```
                                              1.0);
                    break;
            case 2200:
                    for(i = 0; i < M; i++)
                            for(j = 0; j < 30; j++)
                                    pop[(D * i) + j] = RandomDouble(0.0,
                                              1.0);
                    break;
            case 2300:
                    for(i = 0; i < M; i++)
                            for(j = 0; j < 30; j++)
                                    pop[(D * i) + j] = RandomDouble(0.0,
                                              1.0);
                    break;
            case 2400:
                    // 0 <= x1 <= 1
                    for(i = 0; i < M; i++)
                            pop[(D * i) + 0] = RandomDouble(0.0, 1.0);
                    // -5 <= x2...x10 <= 5
                    for(i = 0; i < M; i++)
                            for(j = 1; j < 10; j++)
                                    pop[(D * i) + j] = RandomDouble
                                            (-5.0, 5.0);
                    break;
            case 2600:
                    for(i = 0; i < M; i++)
                            for(j = 0; j < 10; j++)
                                    pop[(D * i) + j] = RandomDouble(0.0,
                                              1.0);
                    break;

        }
}

double chfun(double *pop, unsigned int fun) {
        CONT_FUN++;

        switch (fun) {

        case 100: return veld_1_1(pop);
        case 101: return veld_1_2(pop);
        case 200: return veld_2_1(pop);
        case 201: return veld_2_2(pop);
        case 300: return veld_3_1(pop);
        case 301: return veld_3_2(pop);
        case 400: return veld_4_1(pop);
        case 401: return veld_4_2(pop);
        case 500: return veld_5_1(pop);
        case 501: return veld_5_2(pop);
        case 502: return veld_5_3(pop);
        case 600: return veld_6_1(pop);
        case 601: return veld_6_2(pop);
        case 700: return veld_7_1(pop);
        case 701: return veld_7_2(pop);
        case 702: return veld_7_3(pop);

        case 1100: return dtlz_1_1(pop);
        case 1101: return dtlz_1_2(pop);
        case 1102: return dtlz_1_3(pop);
        case 1200: return dtlz_2_1(pop);
        case 1201: return dtlz_2_2(pop);
        case 1202: return dtlz_2_3(pop);
```

```
        case 1300: return dtlz_3_1(pop);
        case 1301: return dtlz_3_2(pop);
        case 1302: return dtlz_3_3(pop);
        case 1400: return dtlz_4_1(pop);
        case 1401: return dtlz_4_2(pop);
        case 1402: return dtlz_4_3(pop);
        case 1500: return dtlz_5_1(pop);
        case 1501: return dtlz_5_2(pop);
        case 1502: return dtlz_5_3(pop);
        case 1600: return dtlz_6_1(pop);
        case 1601: return dtlz_6_2(pop);
        case 1602: return dtlz_6_3(pop);
        case 1700: return dtlz_7_1(pop);
        case 1701: return dtlz_7_2(pop);
        case 1702: return dtlz_7_3(pop);

        case 2100: return zdt_1_1(pop);
        case 2101: return zdt_1_2(pop);
        case 2200: return zdt_2_1(pop);
        case 2201: return zdt_2_2(pop);
        case 2300: return zdt_3_1(pop);
        case 2301: return zdt_3_2(pop);
        case 2400: return zdt_4_1(pop);
        case 2401: return zdt_4_2(pop);
        case 2600: return zdt_6_1(pop);
        case 2601: return zdt_6_2(pop);
        }

return 0;
}

void keepin(double *pop, double *vel, unsigned int fun, unsigned int M, unsigned int D) {
        unsigned int i, j;
        double linf[D], lsup[D];

        switch (fun) {
                case 100:
                        linf[0] = -100000.0;
                lsup[0] = 100000.0;
                break;
                case 200:
                        for(i = 0; i < D; i++) {
                                linf[i] = -4.0;
                        lsup[i] = 4.0;
                        }
                break;
                case 300:
                        for(i = 0; i < D; i++) {
                                linf[i] = -3.1416;
                        lsup[i] = 3.1416;
                        }
                break;
                case 400:
                        for(i = 0; i < D; i++) {
                                linf[i] = -5.0;
                        lsup[i] = 5.0;
                        }
                break;
                case 500:
                        for(i = 0; i < D; i++) {
                                linf[i] = -3.0;
```

```
                    lsup[i] = 3.0;
                    }
            break;
            case 600:
                    for(i = 0; i < D; i++) {
                            linf[i] = 0.0;
                    lsup[i] = 1.0;
                    }
            break;
            case 700:
                    for(i = 0; i < D; i++) {
                            linf[i] = -4.0;
                    lsup[i] = 4.0;
                    }
            break;
            //DTZL
            case 1100:
                    for(i = 0; i < D; i++) {
                            linf[i] = 0.0;
                    lsup[i] = 1.0;
                    }
            break;
            case 1200:
                    for(i = 0; i < D; i++) {
                            linf[i] = 0.0;
                    lsup[i] = 1.0;
                    }
            break;
            case 1300:
                    for(i = 0; i < D; i++) {
                            linf[i] = 0.0;
                    lsup[i] = 1.0;
                    }
            break;
            case 1400:
                    for(i = 0; i < D; i++) {
                            linf[i] = 0.0;
                    lsup[i] = 1.0;
                    }
            break;
            case 1500:
                    for(i = 0; i < D; i++) {
                            linf[i] = 0.0;
                    lsup[i] = 1.0;
                    }
            break;
            case 1600:
                    for(i = 0; i < D; i++) {
                            linf[i] = 0.0;
                    lsup[i] = 1.0;
                    }
            break;
            case 1700:
                    for(i = 0; i < D; i++) {
                            linf[i] = 0.0;
                    lsup[i] = 1.0;
                    }
            break;

            case 2100:
                    for(i = 0; i < D; i++) {
                            linf[i] = 0.0;
```

```
                        lsup[i] = 1.0;
                        }
                break;
                case 2200:
                        for(i = 0; i < D; i++) {
                                linf[i] = 0.0;
                        lsup[i] = 1.0;
                        }
                break;
                case 2300:
                        for(i = 0; i < D; i++) {
                                linf[i] = 0.0;
                        lsup[i] = 1.0;
                        }
                break;
                case 2400:
                        linf[0] = 0.0;
                        lsup[0] = 1.0;
                        for(i = 1; i < D; i++) {
                                linf[i] = -5.0;
                        lsup[i] = 5.0;
                        }
                break;
                case 2600:
                        for(i = 0; i < D; i++) {
                                linf[i] = 0.0;
                        lsup[i] = 1.0;
                        }
                break;


        }

        for(i = 0; i < M; i++) {
                for(j = 0; j < D; j++) {
                        if (pop[(D * i) + j] < linf[j]) {
                                pop[(D * i) + j] = linf[j];
                                vel[(D * i) + j] = -vel[(D * i) + j];
                        }
                        if (pop[(D * i) + j] > lsup[j]) {
                                pop[(D * i) + j] = lsup[j];
                                vel[(D * i) + j] = -vel[(D * i) + j];
                        }
                }
        }

}
```

Listing A.8: randomlib.h

```
   results compared with the original FORTRAN version.
   April 1989
   Karl-L. Noell <NOELL@DWIFH1.BITNET>
      and Helmut Weber <WEBER@DWIFH1.BITNET>

   This random number generator originally appeared in "Toward␣a␣Universal
␣␣␣Random␣Number␣Generator" by George Marsaglia and Arif Zaman.
   Florida State University Report: FSU-SCRI-87-50 (1987)
   It was later modified by F. James and published in "A␣Review␣of␣Pseudo-
␣␣␣random␣Number␣Generators"
```

```
   THIS IS THE BEST KNOWN RANDOM NUMBER GENERATOR AVAILABLE.
   (However, a newly discovered technique can yield
   a period of 10^600. But that is still in the development stage.)
   It passes ALL of the tests for random number generators and has a period
   of 2^144, is completely portable (gives bit identical results on all
   machines with at least 24-bit mantissas in the floating point
   representation).
   The algorithm is a combination of a Fibonacci sequence (with lags of 97
   and 33, and operation "subtraction plus one, modulo one") and an
   "arithmetic sequence" (using subtraction).

   Use IJ = 1802 & KL = 9373 to test the random number generator. The
   subroutine RANMAR should be used to generate 20000 random numbers.
   Then display the next six random numbers generated multiplied by
      4096*4096
   If the random number generator is working properly, the random numbers
   should be:
           6533892.0  14220222.0  7275067.0
           6172232.0  8354498.0   10633180.0
*/

/* Globals */
double u[97],c,cd,cm;
int i97,j97;
int test = FALSE;

/*
   This is the initialization routine for the random number generator.
   NOTE: The seed variables can have values between:    0 <= IJ <= 31328
                                                        0 <= KL <= 30081
   The random number sequences created by these two seeds are of sufficient
   length to complete an entire calculation with. For example, if sveral
   different groups are working on different parts of the same calculation,
   each group could be assigned its own IJ seed. This would leave each group
   with 30000 choices for the second seed. That is to say, this random
   number generator can create 900 million different subsequences -- with
   each subsequence having a length of approximately 10^30.
*/
void RandomInitialise(int ij,int kl)
{
   double s,t;
   int ii,i,j,k,l,jj,m;

   /*
      Handle the seed range errors
         First random number seed must be between 0 and 31328
         Second seed must have a value between 0 and 30081
   */
   if (ij < 0 || ij > 31328 || kl < 0 || kl > 30081) {
      ij = 1802;
      kl = 9373;
   }

   i = (ij / 177) % 177 + 2;
   j = (ij % 177)       + 2;
   k = (kl / 169) % 178 + 1;
   l = (kl % 169);

   for (ii=0; ii<97; ii++) {
      s = 0.0;
      t = 0.5;
      for (jj=0; jj<24; jj++) {
```

```
         m = (((i * j) % 179) * k) % 179;
         i = j;
         j = k;
         k = m;
         l = (53 * l + 1) % 169;
         if (((l * m % 64)) >= 32)
            s += t;
         t *= 0.5;
      }
      u[ii] = s;
   }

   c    = 362436.0 / 16777216.0;
   cd   = 7654321.0 / 16777216.0;
   cm   = 16777213.0 / 16777216.0;
   i97  = 97;
   j97  = 33;
   test = TRUE;
}

/*
   This is the random number generator proposed by George Marsaglia in
   Florida State University Report: FSU-SCRI-87-50
*/
double RandomUniform(void)
{
   double uni;

   /* Make sure the initialisation routine has been called */
   if (!test)
      RandomInitialise(1802,9373);

   uni = u[i97-1] - u[j97-1];
   if (uni <= 0.0)
      uni++;
   u[i97-1] = uni;
   i97--;
   if (i97 == 0)
      i97 = 97;
   j97--;
   if (j97 == 0)
      j97 = 97;
   c -= cd;
   if (c < 0.0)
      c += cm;
   uni -= c;
   if (uni < 0.0)
      uni++;

   return(uni);
}

/*
  ALGORITHM 712, COLLECTED ALGORITHMS FROM ACM.
  THIS WORK PUBLISHED IN TRANSACTIONS ON MATHEMATICAL SOFTWARE,
  VOL. 18, NO. 4, DECEMBER, 1992, PP. 434-435.
  The function returns a normally distributed pseudo-random number
  with a given mean and standard devaiation.  Calls are made to a
  function subprogram which must return independent random
  numbers uniform in the interval (0,1).
  The algorithm uses the ratio of uniforms method of A.J. Kinderman
  and J.F. Monahan augmented with quadratic bounding curves.
```

```
*/
double RandomGaussian(double mean,double stddev)
{
   double  q,u,v,x,y;

   /*
      Generate P = (u,v) uniform in rect. enclosing acceptance region
      Make sure that any random numbers <= 0 are rejected, since
      gaussian() requires uniforms > 0, but RandomUniform() delivers >= 0.
   */
   do {
      u = RandomUniform();
      v = RandomUniform();
      if (u <= 0.0 || v <= 0.0) {
          u = 1.0;
          v = 1.0;
      }
      v = 1.7156 * (v - 0.5);

      /*  Evaluate the quadratic form */
      x = u - 0.449871;
      y = fabs(v) + 0.386595;
      q = x * x + y * (0.19600 * y - 0.25472 * x);

      /* Accept P if inside inner ellipse */
      if (q < 0.27597)
         break;

      /*  Reject P if outside outer ellipse, or outside acceptance region */
    } while ((q > 0.27846) || (v * v > -4.0 * log(u) * u * u));

    /*  Return ratio of P's coordinates as the normal deviate */
    return (mean + stddev * v / u);
}

/*
   Return random integer within a range, lower -> upper INCLUSIVE
*/
int RandomInt(int lower, int upper)
{
   return((int)(RandomUniform() * (upper - lower + 1)) + lower);
}

/*
   Return random float within a range, lower -> upper
*/
double RandomDouble(double lower, double upper)
{
   return((upper - lower) * RandomUniform() + lower);
}
```